Universal Inter-Blockchain Protocol (UIP)

ABSTRACT

Blockchain interoperability, that is, allowing state transitions across different blockchain networks, would facilitate more significant blockchain adoption. However, existing protocols are neither general enough to natively interconnect arbitrary blockchain networks, nor secure enough to prove the correctness of inter-chain operations, causing today's blockchain networks to be isolated islands. In this paper, we present UIP, the first *generic, secure and trustfree* inter-chain protocol that enables true blockchain interoperability. UIP is (i) generic, operating on any blockchain with a public transaction ledger, (ii) secure, by which we mean that each inter-chain operation either finishes with verifiable correctness or aborts due to security violations (where misbehaving entities can be held accountable inside a blockchain), and (iii) financially safe, meaning that applications of UIP can be made financially atomic, regardless of how or where the applications terminate. We rigorously prove UIP's security guarantee using the UC-framework. To further evaluate UIP's functionality, we build a prototype platform containing four independent blockchain networks with distinct consensus protocols, upon which we implement two categories of decentralized applications using UIP. Through our experiments, we demonstrate the feasibility of UIP in practice and show that these UIP-powered applications advance the state-of-the-art blockchain applications.

1 INTRODUCTION

Blockchain is a distributed ledger that provides a decentralized consensus protocol to validate transactions among peers inside a network. Over the last few years, we have witnessed rapid growth of several flagship blockchain applications, such as the payment system Bitcoin [42] and the smart contract platform Ethereum [15]. Since then, considerable amount of effort has been made to improve the performance of a blockchain network, such as more efficient consensus algorithms [5, 11, 22, 32], improving transaction rate by sharding [13, 33, 40, 50] and payment channels [25, 29, 41], enhancing the privacy for smart contracts [18, 28, 34], and reducing their vulnerabilities via program analysis [14, 35, 39].

As a result, in today's blockchain ecosystem, we see the proliferation of many distinct blockchains, falling roughly into the categories of public, private and consortium blockchains [3]. Each of them enters the market by claiming certain improvement on a specific property of blockchain. However, today's blockchain networks are almost-isolated islands, disallowing easy and efficient state transitions across different blockchain networks. The lack of interoperability limits users to a single chosen blockchain, preventing them from embracing the performance improvements made on other blockchains.

Moreover, the lack of interoperability may even hinder the growth and widespread adoption of blockchain. In retrospect, one significant reason that enabled the growth of the Internet from a small research network into a core part of the global economy is its domain-driven architecture built upon scalable inter-domain routing protocols. This architecture allows each domain (referred to as an Autonomous System) to evolve independently (*e.g.,* deploying proprietary protocols such as SDN [26, 27] and virtualization [44]), and interconnects those domains using a general interdomain protocol (specifically, the Border Gateway Protocol [16, 45]) to deliver end-to-end connectivity for users. In the blockchain ecosystem, although intensive research has been done on improving each individual blockchain, few efforts have been made to establish an interoperability protocol.

We recognize at least three major challenges in this regard. First, the interoperability protocol should be *universally applicable, such that it facilitates interoperation among various types of underlying blockchain networks.* Two existing notable protocols, Cosmos [4] and Polkadot [8], fail to deliver such generality because they are designed primarily for interconnecting their proprietary blockchains Parachains and Tindermint, respectively.

Second, a blockchain network is generally abstracted as a persistent state machine whose correctness is ensured by its consensus protocol. However, it is difficult to ensure the correctness of multi-chain state transitions since executions on different blockchains are *not* synchronized. This problem becomes even more challenging when state updates on different blockchains have dependency requirements, since relying on a trusted authority for coordination would significantly limit the applicability of the interoperability protocol. This challenge alone demonstrates that blockchain interoperability is not a simple reduction of the Internet interconnection protocols, which just deliver stateless reachability.

Third, blockchain is touted as a technology for revolutionizing the financial sector [43, 47], which usually demands *stricter security requirements, including financial safety, transaction integrity, and accountability.* However, none of the existing interoperability proposals are designed with well-formulated security properties. Given the immutable nature of blockchains, this lack of protocol security significantly limits the use of cross-chain applications in the financial sector.

In this paper, we present UIP (short for universal inter-blockchain protocol) that simultaneously offers *generic*, *secure* and *financially safe* inter-blockchain capability. To ensure these properties, UIP consists of three novel components that work seamlessly together: Verifiable Execution System (VES), Network Status Blockchain (NSB), and Insurance Smart Contract (ISC).

The VES is a trust-free service that executes transactions across several blockchains on behalf of a decentralized application (dApp). The VES takes a cross-chain operation given by the dApp, and decomposes it into a transaction dependency graph, which contains a set of transactions and their precondition and deadline requirements. At runtime, if the transaction executions by the VES and dApp comply with these preconditions and deadlines, the operation will finish correctly as desired by the dApp.

Since no mutual trust is required between the dApp and VES, the NSB and ISC are therefore designed to enforce the operation correctness in a trust-free manner. The NSB essentially is a *blockchain of blockchains* designed to provide an objective view on the operation

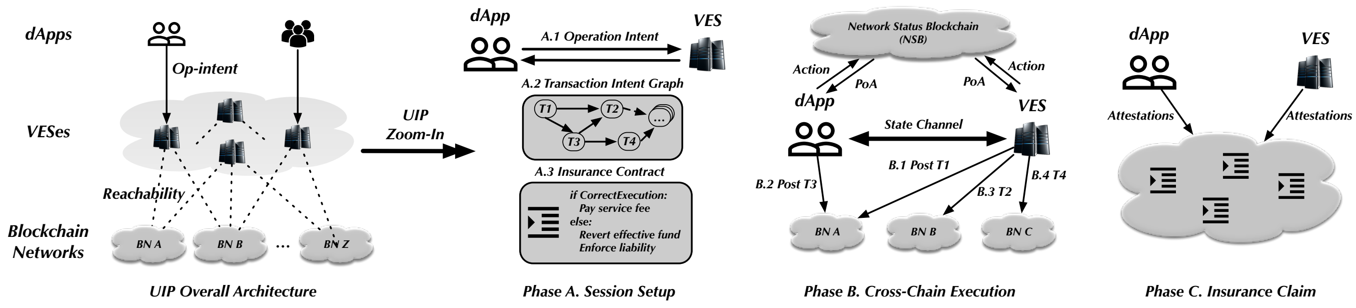


Figure 1: The architecture of UIP and the high-level execution process of an Op.

status, where the objectivity is guaranteed by Merkle proofs. In particular, the NSB enables dApp and VES to construct Merkle proofs to prove the status of each transaction in the transaction graph, and their actions taken while executing the transaction graph. The ISC then takes the Merkle proofs reported by both parties to decide the final status of the operation. The ISC claims operation correctness only if it receives sufficient proofs to demonstrate that no precondition and deadline rules are violated. Otherwise, it claims failure by *financially reverting* all executed transactions and penalizing the entity that is accountable for the failure. As a prerequisite, both the dApp and VES are required to stake funds with the ISC as *indemnity deposits*, based on which the ISC can ensure the above financial atomicity and enforce accountability.

2 UIP OVERVIEW

UIP is a protocol that securely realizes inter-blockchain executions. Decentralized applications (dApps) that need inter-chain operations employ UIP to achieve multi-blockchain state updates. Since a dApp's state transitions are driven by a set of transactions executed on multiple blockchain networks (BNs), UIP, without loss of generality, represents an inter-BN operation as $(\mathcal{A}, out) = \text{Op}(\mathcal{G}_T, \text{ISC}, \text{VES})$. Here, Op represents the inter-BN operation that conducts a set of transactions on multiple BNs. $\mathcal{G}_T$ represents the transaction set as a dependency graph, which specifies the preconditions and deadlines of each transaction to ensure the correctness of Op. The actual execution of Op involves the participation of a Verifiable Execution System (VES) selected by dApp. Since no mutual trust between dApp and VES is required, an insurance contract ISC is added to minimize risk. The outputs of Op include an optional dApp-related output *out*, and a collection of security attestations $\mathcal{A}$, based on which the ISC and interested third parties can verify the correctness or violation of Op.

2.1 UIP Architecture

Figure 1 depicts the architecture of UIP and the high-level inter-BN execution process through UIP. UIP involves three types of entities: dApps that request inter-BN operations, Verifiable Execution Nodes (VESes) that drive the actual executions across different BNs, and Insurance Smart Contracts (ISCs) that arbitrate the correctness or violation of Op and guarantees financial safety for dApps and VESes in case of exceptions. As a security-oriented optimization, UIP further designs a Network Status Blockchain (NSB) to provide an objective view of the execution status of Op, allowing UIP (and ISCs) to unambiguously determine the faulty entity in case Op does not run to completion. The entire execution process of Op has three phases. We next explain the entities and the execution process.

2.1.1 *Clients of UIP: dApps*

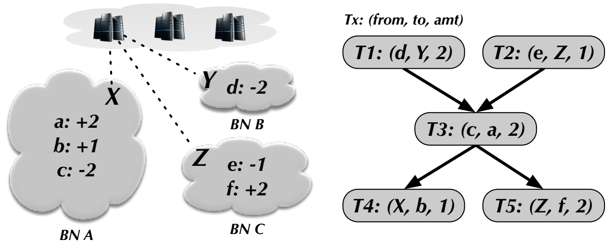
UIP is a platform architecture for dApps built on inter-BN operations. UIP employs an intent-driven design in which a dApp requests an inter-BN operation by expressing an *op-intent*, which is an abstract declaration of what the dApp desires to achieve through UIP, such as an inter-BN payment or a sequential cross-BN smart contract invocation. Figure 2(a) shows an example of a group payment op-intent requested by our Atomizer dApp (see details in § 4.1). This op-intent specifies the final balance change for a group of dApp accounts (referred to as $\boldsymbol{a}$–$\boldsymbol{f}$) on multiple BNs. Upon receiving the op-intent, the VES is responsible for computing the transaction dependency graph $\mathcal{G}_T$, as described in § 2.1.2 below.

We clarify that a dApp may further have its own clients who are the actual beneficiaries of Op. For instance, in Atomizer, the accounts sending or receiving funds are actually owned by the clients of Atomizer. For simplicity in describing UIP, we conceptually merge the end clients and the dApp into a single unified party that holds multiple accounts. However, the ISC can revert funds to specific dApp clients if the execution of Op aborts prematurely.

2.1.2 *Verifiable Execution Systems*

A VES is a distributed system that drives the cross-BN executions upon receiving op-intents from dApps. Each VES defines its own service model, including reachability (*i.e.,* the set of BNs that the VES supports), service fee charged for correct executions, and insurance plan (*i.e.,* the expected compensation to the dApp if VES's execution is proven incorrect). dApps have full autonomy to select VESes that match their operation requirements. In Section § A.4, we discuss the anatomy of VESes.

The first phase of Op is the session setup where a VES computes a dependency graph $\mathcal{G}_T$ that *fulfills* the Op as follows. The vertices of $\mathcal{G}_T$ are referred to as *transaction intents* that define the actions required by both VES and dApp to execute Op. Each transaction intent is specifically and programmatically linked to an executable *on-chain transaction* sent by either VES or dApp in the second phase of Op. We differentiate a transaction intent with its on-chain counterpart because the computation of the on-chain transaction may be infeasible in the first phase (*c.f.,* § 2.2.1). We say that $\mathcal{G}_T$ fulfills Op if the commitment of all transaction intents in $\mathcal{G}_T$ (precisely speaking, their on-chain counterparts) results in a multi-BN state that is consistent with dApp's op-intent. For example, the vertices



(a) Group Payment Op-intent *(b) Transaction Dependency Graph*

Figure 2: (a) An inter-BN group payment op-intent requested by the Atomizer dApp; (b) the corresponding $\mathcal{G}_T$ generated by a VES to fulfill the op-intent.

in Figure 2(b) are the transaction intents proposed by the VES to fulfill the group payment op-intent in Figure 2(a). ***X*–*Z*** are *relay accounts* created by the VES on each involved BN to match the balance updates of dApp's accounts.

However, having a set of unordered transaction intents alone may be insufficient to ensure the correctness of Op. For instance, in the above group payment example, to achieve external consistency [24], dApp accounts that receive payments should not see balance increases before the paying accounts reduce their balances. In our HyperService dApp discussed in § 4.2, a transaction $\mathcal{T}_2$ may take as input the *output* of another transaction $\mathcal{T}_1$ that is sent to a remote oracle contract (the output of $\mathcal{T}_1$ is typically represented by a state update on the oracle contract). In this case, $\mathcal{T}_1$ is a strict precondition of $\mathcal{T}_2$. Therefore, the correctness of Op not only requires all transaction intents in $\mathcal{G}_T$ to be committed, but also imposes constraints on the order in which they are committed.

To enforce the ordering constraints, UIP defines the correctness model of Op as a *Transaction Dependency Graph* that specifies the *preconditions* and deadlines on these transactions. Under this correctness model, a transaction intent can proceed only if all its dependent intents are finalized, and a transaction intent must proceed within a bounded period of time after its dependencies are satisfied. In the $\mathcal{G}_T$ example shown in Figure 2(b), T_3 should be executed only after both T_1 and T_2 are committed on their respective BNs. The exact deadline definition is discussed in § 2.2.2.

Our generic correctness model provides two desirable properties. First, imposing preconditions on cross-BN executions enables UIP to navigate a dApp from its source state to its destination state via dApp-defined "routing", even though transactions executed on different BNs are not synchronized. Second, adding deadlines is not only crucial to achieve timely cross-BN executions (important for operations that are time sensitive, such as token exchanges), but also critical to ensure forward progress, preventing malicious entity from taking an arbitrary amount of time to perform the next step of the Op.

The correctness model of Op is not unique. The model employed by UIP emphasizes *generality* in distributed computing where events are often (partially) ordered to honor dependency among events [36]. UIP can be extended with dApp-specific correctness requirements as necessary.

2.1.3 Insurance Smart Contract

We now discuss how UIP enforces the above correctness model. In particular, the correctness of Op is enforced such that if both parties execute Op correctly (*i.e.,* all preconditions and deadlines are satisfied), then such correctness is publicly verifiable by any interested entities; if Op aborts prematurely due to rule violations, UIP holds the misbehaving entities accountable, and financially reverts all committed transactions in $\mathcal{G}_T$ to ensure financial safety.

Enforcing the correctness of Op without introducing trusted authorities raises another challenge in UIP's design. UIP addresses this using *security attestations* and the Insurance Smart Contract ISC. Specifically, attestations are publicly verifiable security proofs that certify the status of each transaction intent in $\mathcal{G}_T$. Both parties compute and publish attestations during the execution phase of Op, possibly relying on the view provided by the NSB (introduced below). The ISC then takes attestations as input and decides the final status of Op. As shown in § 3.2, UIP ensures no ambiguity is raised in the correctness enforcement.

The ISC is created and deployed on a BN mutually agreed by both parties in the first phase of Op. Afterwards, it is invoked by both parties using attestations and expires after both parties claim their insurance in the last phase of Op.

2.1.4 Network Status Blockchain

The NSB is a crucial security extension to UIP. Conceptually, it is a *blockchain of blockchains* that provides an objective view of the execution status of Op. On one hand, the NSB consolidates a unified point-of-view of each underlying BNs' transactions and transaction pools, providing an objective and trust-free environment for monitoring the transaction status for Op. On the other hand, the NSB enables Proofs of Actions (PoAs) for both the dApp and VES, ensuring that the ISC has sufficient information to decide which party to blame if Op aborts incorrectly. In this paper, we do not assume that NSB is *perfect*, *i.e.,* it offers unbounded throughput and instant finality. In fact, an NSB with performance comparable to that of today's blockchains is sufficient to support the aforementioned security extensions. In addition, we envision that the NSB's functionality is extensible to host generic decentralized services that can improve the overall usability of UIP. For instance, as shown in § 2.2.4, an immediate use case is using NSB as a decentralized discrete timer to provide authoritative timestamps, which simplifies the enforcement of transaction deadlines.

2.2 Protocol Overview

In this section, we provide the protocol overview for all three phases shown in Figure 1. The full protocol details are specified in § 3. For simplicity of protocol description, we consider an example where a dApp $\mathcal{D}$ selects a VES $\mathcal{V}$ for Op.

2.2.1 Session Setup for Op

Two major goals of the session setup phase are computing the transaction dependency graph $\mathcal{G}_T$ and deploying the insurance contract ISC. The actual $\mathcal{G}_T$ computation depends on the application context. We first discuss the format of $\mathcal{G}_T$ in this section, and defer the detailed $\mathcal{G}_T$ computation to § 4, where concrete application contexts are given for our Atomizer and HyperService dApps.

Since each BN has its own transaction format, to be generic, UIP represents a transaction intent in $\mathcal{G}_T$ as $\mathcal{T}$:= [from, to, seq, meta], where the pair <from, to> decides the sending and receiving addresses of $\mathcal{T}$, seq represents the sequence number of $\mathcal{T}$ in $\mathcal{G}_T$, and meta stores the structured and customizable metadata associated with $\mathcal{T}$. In order to achieve financial atomicity, meta must populate an amt field to specify the *Net Balance Gain* of $\mathcal{T}$, which represents the value of funds transferred from the *from* address to the *to* address by $\mathcal{T}$. We discuss how to decide the value of amt in § 2.2.3.

Linkability between transaction intents in $\mathcal{G}_T$ with their corresponding on-chain counterparts is a prerequisite for the ISC to insure the execution of $\mathcal{G}_T$. In general, to achieve this linkability, the <from, to> pair in $\mathcal{T}$ should be consistent with the pair specified in $\mathcal{T}$'s corresponding on-chain transaction $\widetilde{T}$. Since the dApp could require the VES to use freshly generated addresses for each new session, the address pairs of $\mathcal{T}$ and $\widetilde{T}$ can provide high confidence of linkability. In addition, both parties can further rely on a customizable field in meta for linkability, as long as that the field is programmatically recognizable by ISC. For instance, if the on-chain transaction $\widetilde{T}$ can be decided *a priori* when computing $\mathcal{G}_T$, or if it is possible to embed a cookie into $\widetilde{T}$, both parties could add an optional field fingerprint in meta to specify a pre-agreed identifier that is uniquely linkable with $\widetilde{T}$.

In the case where it is impossible to decide $\widetilde{T}$ *a priori* (*e.g.,* in the HyperService dApp shown in § 4.2, $\widetilde{T}$ may depend on the resulting state of its pre-conditioned transactions), the ISC is informed of such state when processing the attestations sent by both parties, so that it can correctly verify the linkability between $\widetilde{T}$ and $\mathcal{T}$ (details are given in § 3.2). We acknowledge that certain privacy blockchain features, such as anonymous double-spending protection [48], may pose additional challenges on linkability, but these issues are beyond the scope of this paper.

Both parties should mutually agree on $\mathcal{G}_T$ before proceeding to the execution phase of Op. The correctness check includes (i) the commitment of all transactions in $\mathcal{G}_T$ fulfills the op-intent given by $\mathcal{D}$, and the preconditions among transactions are as desired and loop-free; (ii) the meta of each transaction intent is correctly populated with proper amt, linkability capability, and some customized configurations (*e.g.,* a customized deadline) if specified. Once $\mathcal{G}_T$ is mutually agreed, $\mathcal{D}$ and $\mathcal{V}$ further create and deploy an ISC before entering the second phase. As ISC takes input as attestations, we defer its design detail in § 2.2.3 after discussing how attestations are computed in the second phase of Op.

2.2.2 *Cross-BN Execution of Op*

In this section, we present the *base protocol* for the cross-BN execution phase of Op to provide the reader with a clear blueprint. We then clarify multiple technical subtleties and propose extensions atop the base protocol. In § 3.2.1 and § 3.2.2, we specify the fully-detailed execution protocol between $\mathcal{D}$ and $\mathcal{V}$.

We define the *status* of a transaction intent to be *unknown*, *open*, or *closed*. Each transaction intent is initialized as the *unknown* status. In the execution phase, a transaction intent $\mathcal{T}$ becomes eligible for opening if each precondition of $\mathcal{T}$ has been *finalized* with correct proofs. A transaction intent $\mathcal{T}$ becomes closed at a predefined stage in $\widetilde{T}$'s lifecycle. For example, the VES and dApp could define an intent as closed as soon as its on-chain counterpart $\widetilde{T}$ is added to the pending transaction pool of $\widetilde{T}$'s BN, or alternatively, after $\widetilde{T}$ has reached a certain level of finality within its BN. Without loss of generality, in this section, we describe the base protocol using the latter definition. § A.2 elaborates on the other definition. Regardless of the closure definition, a *deadline* defines the latency bound from the *open* status to the *closed* status of a transaction intent. This constraint is later enforced by the ISC.

1. $\mathcal{V}$ and $\mathcal{D}$ mutually agree on a session ID sid for Op, as well as a session key used to encrypt all messages exchanged between them via the off-chain channels.
2. $\mathcal{V}$ determines, based on the current state of $\mathcal{G}_T$, the set of transaction intents that are eligible to be opened. In this example, intent $\mathcal{T}_1$ is processed first.
3. $\mathcal{V}$, the originator of $\mathcal{T}_1$, computes the on-chain transaction $\widetilde{T}_1$ for $\mathcal{T}_1$, and then it sends $\mathcal{D}$ a certificate C_{pre} to initiate the opening of $\widetilde{T}_1$, where $C_{\text{pre}} := \text{Cert}([\widetilde{T}_1, \text{init}, \text{sid}, \mathcal{T}_1]; \text{Sig}^{\mathcal{V}}_{\text{sid}})$.
   Notation: $\text{Cert}([*]; \text{Sig}^{\mathcal{V}}_{\text{sid}})$ represents a certificate proving that $\mathcal{V}$ agrees to some value by signing over the value.
4. Upon receiving C_{pre}, $\mathcal{D}$ verifies $\widetilde{T}_1$ is correctly *associated* with $\mathcal{T}_1$, where association is precisely defined below. Then it replies to $\mathcal{V}$ with a timestamped $C_{\text{ack}} := \text{Cert}([\widetilde{T}_1, \text{open}, \text{sid}, \mathcal{T}_1, \text{ts}_{\text{open}}]; \text{Sig}^{\mathcal{D}}_{\text{sid}})$;
5. If $\mathcal{V}$ accepts the opening timestamp ts_{open} added in C_{ack}, it appends C_{ack} with its own signature to produce a dual-signed attestation $\text{Atte}^{o}_{\mathcal{T}_1} := \text{Cert}([\widetilde{T}_1, \text{open}, \text{sid}, \mathcal{T}_1, \text{ts}_{\text{open}}]; \text{Sig}^{\mathcal{V}}_{\text{sid}}, \text{Sig}^{\mathcal{D}}_{\text{sid}})$. Now it is safe for $\mathcal{V}$ to dispatch $\widetilde{T}_1$ for on-chain execution.
6. Once $\widetilde{T}_1$ is finalized on its destination BN, $\mathcal{V}$ and $\mathcal{D}$ go through another propose-ack negotiation to generate the closing attestation $\text{Atte}^{c}_{\mathcal{T}_1}$ to conclude the execution of $\mathcal{T}_1$, where $\text{Atte}^{c}_{\mathcal{T}_1} := \text{Cert}([\widetilde{T}_1, \text{closed}, \text{sid}, \mathcal{T}_1, \text{ts}_{\text{closed}}]; \text{Sig}^{\mathcal{V}}_{\text{sid}}, \text{Sig}^{\mathcal{D}}_{\text{sid}})$.
7. After processing $\text{Atte}^{c}_{\mathcal{T}_1}$, $\mathcal{V}$ updates its internal bookkeeping and then computes a new set of transaction intents that are eligible to be opened based on the updated $\mathcal{G}_T$. In this example, $\mathcal{T}_2$ are eligible to be opened.
8. At any time before ISC expiration, both parties can invoke ISC with attestations $\mathcal{A} := \{\text{Atte}^{o}_{\mathcal{T}_1}, \text{Atte}^{c}_{\mathcal{T}_1}, \text{Atte}^{o}_{\mathcal{T}_2}, \text{Atte}^{c}_{\mathcal{T}_2}\}$.

Figure 3: The base protocol for inter-BN execution.

Each transaction intent $\mathcal{T}$ is executed in three stages (open $\longrightarrow$ dispatch $\longrightarrow$ close). In the open stage, $\mathcal{D}$ and $\mathcal{V}$ together compute an *open* attestation to prove that $\mathcal{T}$ is eligible to be opened. Both parties proceed with $\mathcal{T}$ only if the open stage completes successfully. In the dispatch stage, the originator of $\mathcal{T}$ (either $\mathcal{D}$ or $\mathcal{V}$) computes the on-chain counterpart $\widetilde{T}$ for $\mathcal{T}$ and posts $\widetilde{T}$ on its destination BN for on-chain execution. In the close stage, $\mathcal{D}$ and $\mathcal{V}$ together compute a *closed* attestation to prove that $\mathcal{T}$ has finished executing. In the base protocol, both parties negotiate open and close stages through off-chain state channels. The full protocol (discussed in § 3.2) defines an additional on-chain negotiation mechanism using the NSB, which offers stronger security guarantees. As an example, Figure 3 summarizes the execution steps for an Op whose $\mathcal{G}_T$ contains two transaction intents $\mathcal{T}_1$ and $\mathcal{T}_2$. Without loss of generality, we assume $\mathcal{T}_1$ originates from $\mathcal{V}$.

Technical Subtleties. We clarify several subtleties in the base protocol. First, in step (3), if $\mathcal{D}$ is the originator of $\mathcal{T}_1$, $\mathcal{V}$ needs to send an initialization request to $\mathcal{D}$ to start the process. The rationale behind this design is to release $\mathcal{D}$ from needing to proactively monitor all underlying BNs involved in Op. Instead, the protocol always relies on $\mathcal{V}$ to dynamically compute the new set of transactions that are eligible to be opened whenever a transaction in $\mathcal{G}_T$ is closed. Thus, $\mathcal{V}$ is the party that triggers the processing of

1 Instantiate the ISC with proper configuration, including the session ID sid, public keys $\{\mathsf{pk}^{\mathsf{d}}_{\mathsf{sid}}, \mathsf{pk}^{\mathsf{v}}_{\mathsf{sid}}\}$ for verifying attestations, the contract expiration timer timer_ and so on.

2 Both $\mathcal{V}$ and $\mathcal{D}$ (possibly its actual clients) stake sufficient funds into the ISC to cover each other's losses in case of abnormal termination.

3 The ISC maintains two maps: (i) a transaction state map $\mathsf{T}_{\mathsf{state}} : \mathsf{tid} \to [\mathsf{status}, \mathsf{ts}_{\mathsf{open}}, \mathsf{ts}_{\mathsf{closed}}]$ to record the status of each transaction in $\mathcal{G}_T$; (ii) a fund-reversion map $\mathsf{A}_{\mathsf{revs}} : \mathsf{tid} \to [\mathsf{amt}, \mathsf{dest}]$ to determine the payout to dest if Op, as a whole, fails but tid is committed.

4 Upon receiving an input attestation $\mathsf{Atte}^{\{o,c\}}_{\mathsf{tid}}$, the ISC checks its validity. If valid, it inserts or updates $\mathsf{T}_{\mathsf{state}}[\mathsf{tid}]$ accordingly. If tid is closed by the attestation, the ISC updates $\mathsf{A}_{\mathsf{revs}}$ accordingly.

5 The ISC executes the contract term when timer_ expires. The following are some example contract terms:

(a) For each closed transaction, ISC pays service fee to $\mathcal{V}$;

(b) If Op did not complete correctly, for each closed transaction intent $\mathcal{T}$, the ISC evaluates $[\mathsf{amt}, \mathsf{dest}] = \mathsf{A}_{\mathsf{revs}}[\mathcal{T}]$ and compensates dest by amt using the staked funds from the originator of $\mathcal{T}$.

(c) The ISC returns any remaining staked funds.

Figure 4: Overview of the ISC.

all eligible transactions. Thus, if $\mathcal{T}_1$ is self-originated, $\mathcal{V}$ sends C_{pre} enclosing the signed $\widetilde{T}_1$ to $\mathcal{D}$ (as shown in Figure 3); otherwise, $\mathcal{V}$ sends a request $C_{\mathsf{init}} := \mathsf{Cert}([\mathcal{T}_1, \mathsf{init}, \mathsf{sid}]; \mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}})$ to $\mathcal{D}$, who is capable of computing and signing $\widetilde{T}_1$.

Second, in step (4), upon receiving C_{pre}, $\mathcal{D}$ needs to verify that $\widetilde{T}_1$ is properly *associated* with the transaction intent $\mathcal{T}_1$. The association is precisely defined as follows. (i) $\widetilde{T}_1$ and $\mathcal{T}_1$ are linkable, either through the <from, to> address pair or a unique fingerprint field added to meta. (ii) If the computation of $\widetilde{T}_1$ depends on the resulting state of its preconditioned transactions, $\mathcal{D}$ verifies that the state is correct, and the computation of $\widetilde{T}_1$ has correctly incorporated the state. In general, the state verification requires $\mathcal{D}$ to examine the ledgers of underlying BNs. Alternatively, $\mathcal{D}$ may read status from the NSB which provides a unified view of all BNs (*c.f.,* § 2.2.4).

Third, in step (5), $\mathcal{V}$ should make sure that timestamp $\mathsf{ts}_{\mathsf{open}}$ added by $\mathcal{D}$ in C_{ack} is valid. On one hand, $\mathsf{ts}_{\mathsf{open}}$ must be in the format recognizable by the ISC; otherwise $\mathsf{Atte}^{o}_{\mathcal{T}_1}$ would be invalid as an input to the ISC. One the other hand, $\mathsf{ts}_{\mathsf{open}}$ should reasonably reflect the actual time at which $\mathcal{D}$ produces C_{ack}. This is to prevent $\mathcal{D}$ from gaming the protocol by adding a carefully crafted $\mathsf{ts}_{\mathsf{open}}$ (*e.g.,* a future timestamp) to make it harder for $\mathcal{V}$ to satisfy the deadline of $\mathcal{T}_1$. Similarly in step (6), although $\mathcal{V}$ may have a different view from $\mathcal{D}$ as to when $\widetilde{T}_1$ is finalized on its destination BN, $\mathcal{D}$ should compute $\mathsf{Atte}^{c}_{\mathcal{T}_1}$ only if the time difference is within a reasonable range. In order to achieve a more definitive bound of time difference between the two parties, they can obtain *calibrated* timestamps from the NSB, as discussed in § 2.2.4.

Finally, in step (6), the base protocol relies on mutual cooperation between $\mathcal{D}$ and $\mathcal{V}$ to compute $\mathsf{Atte}^{c}_{\mathcal{T}_1}$. As a result, if $\mathcal{T}_1$ is not properly closed (*i.e.,* $\mathsf{Atte}^{c}_{\mathcal{T}_1}$ is missing), that failure could be either because $\mathcal{V}$ fails to dispatch $\widetilde{T}_1$ for on-chain execution in step (5) or because $\mathcal{D}$ fails to cooperate in step (6). The base protocol therefore does not provide enough information to unambiguously decide which party to blame if Op does not finish correctly. This is addressed by our full protocols in § 3.2.

2.2.3 *The Insurance Contract (ISC) of Op*

The ISC takes as input attestations from both parties, based on which it updates its internal state. When the ISC times out, it executes the contract terms based on its internal state, after which its funds are depleted and the contract never runs again. We overview the ISC in Figure 4, and clarify several technical subtleties below.

ISC Initialization and Configuration. In the initialization step, the ISC should be instantiated with proper configuration that is necessary to settle contract terms (some example contract terms are given in step (5) of Figure 4). First and foremost, the ISC must be installed with the arbitration logic for deciding the final status of Op, as well as identifying the accountable entities if Op fails. The high level design is that the ISC decides the status of Op base on the statuses enclosed in all received attentions, and identifies the accountable entities using a decision tree built based on the *proofs of actions* submitted by $\mathcal{D}$ and $\mathcal{V}$. The details are given in § 3.2.3. In addition, the ISC also needs to know the addresses used for accepting the reversion funds if Op fails. These addresses, living on the BN that hosts the ISC, are not necessarily the same as the addresses specified by the transaction intents in $\mathcal{G}_T$. Moreover, both parties may define customized terms for certain transactions in $\mathcal{G}_T$, such as customized deadlines and service fees. Both parties should make sure that the ISC is instantiated as desired before proceeding to the execution phase of the Op.

Fund Staking. As a prerequisite for executing contract terms, both parties must stake sufficient funds in the ISC. Intuitively, to ensure that the ISC holds sufficient funds to financially revert all committed transactions regardless of the step at which Op aborts, each entity (the VES or a client behind dApp) should stake at least the total amount of incoming funds to the entity *without* deducting the outgoing amount. This strawman design, however, require more stakes from the entities. More desirably, considering the dependency requirements in $\mathcal{G}_T$, an entity $\mathcal{X}$ only stakes at least

$$\max_{s\in\mathcal{G}_S} \sum_{\mathcal{T}\in s\ \wedge\ \mathcal{T}.\mathsf{to}=\mathcal{X}} \mathcal{T}.\mathsf{meta.amt} - \sum_{\mathcal{T}\in s\ \wedge\ \mathcal{T}.\mathsf{from}=\mathcal{X}} \mathcal{T}.\mathsf{meta.amt}$$

where $\mathcal{G}_S$ is the set of all committable subsets in $\mathcal{G}_T$, where a subset $s \subseteq \mathcal{G}_T$ is *committable* if, whenever $\mathcal{T} \in s$, all preconditions of $\mathcal{T}$ are also in s. For clarity of notation, throughout the paper, when saying $\mathcal{T}.\mathsf{from} = \mathcal{V}$, we mean $\mathcal{T}$ originates from an account owned by $\mathcal{V}$, and likewise for $\mathcal{D}$. Some additional funds may be required, for instance, to cover violation penalties (from both parties) and service fee (from $\mathcal{D}$). After contract terms are executed, the ISC returns any remaining funds back to their sending entities.

Financial Atomicity. The Net Balance Gain (NBG) amt added in each transaction intent is necessary for financial atomicity. When determining amt for $\mathcal{T}$, $\mathcal{V}$ and $\mathcal{D}$ should take into consideration the exchange rate between the token used for executing $\mathcal{T}$ on-chain and the token used in the ISC. Thus, the financial atomicity might be affected by the fluctuation of token exchange rates. This issue can be mitigated by staking and compensating using stable coins. Extending ISCs to use stable coins for financial atomicity is left as future work.

2.2.4 *Network Status Blockchain*

Block Format. As summarized in § 2.1.4, the NSB provides an objective view of the transaction status of underlying BNs and the

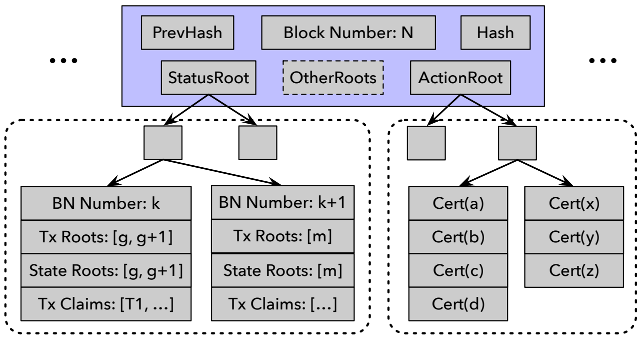


Figure 5: The format of a block on the NSB.

status of actions taken by both dApps and VESes throughout the Op execution. To enable such functionalities, besides the common fields contained by a block of typical blockchains (such as the hash fields to link blocks and the transaction Merkle tree root), we design two additional Merkle tree roots StatusRoot and ActionRoot for an NSB block, as depicted in Figure 5.

StatusRoot is the root of a lexicographically sorted Merkle tree whose leaves store the transaction status of underlying BNs. To support both transaction closure definitions, the NSB represents the transaction status of a BN as three lists: the *Tx Roots*, *State Roots*, and *Tx Claims*. *Tx Roots* and *State Roots* are the lists of transaction Merkle roots and state Merkle roots, respectively, retrieved directly from the BN (*e.g.,* the tx roots and state roots on Etheruem blocks [49]). The *Tx Claims* is a list of transactions claimed by VESes and dApps during the execution of Op. It is required to support the alternative transaction closure definitions where transactions are considered closed once they are added to the transaction pending pools of their destination BNs. The BN-status Merkle tree is sorted to achieve quick element search among leaves, as well as convenient proof of absence (to prove the absence of x, Merkle proofs for the two elements adjacent to x suffice). § A.2 provides a detailed explanation on how the StatusRoot is constructed

ActionRoot is the root of a sorted Merkle tree whose leaf nodes store certificates computed by both parties. Each certificate represents a certain step taken by either $\mathcal{V}$ or $\mathcal{D}$ during the execution of Op; for instance the C_{pre} in step (3) of Figure 3 represents that $\mathcal{V}$ has initiated the opening of $\mathcal{T}_1$. To prove such an action, $\mathcal{V}$ would need to construct a Merkle proof to demonstrate that the C_{pre} has been included in the NSB. The rational behind PoAs is to allow each party to prove to the ISC that it has taken certain steps during execution, so that the ISC is able to decide which party misbehaved if Op aborts prematurely.

We elaborate on how exactly the NSB is utilized in UIP when specifying the full protocol in § 3.2. Here, we list a few usage cases. First, both parties may use the most recent committed block number on the NSB as ts_{open} and $\text{ts}_{\text{closed}}$ timestamps when computing attestations. This provides both parties a unified timing reference. Second, each party can by itself compute a closing attestation for a transaction intent $\mathcal{T}$ by constructing two Merkle proofs: (i) a hash path to prove that $\widetilde{T}$ (the on-chain counterpart of $\mathcal{T}$) is included in a Merkle root R on its destination BN, and (ii) another hash path to prove that R is recorded under one StatusRoot of the NSB. The resulting state by $\widetilde{T}$ is *optional* in the closing proof: it is required only if the ISC needs this state for executing contract terms. To accept Merkle proofs as input, it is preferred that the ISC is deployed on the NSB such that the ISC can verify, within the NSB, that the roots of these proofs are indeed on-chain, rather than having to rely on external trusted oracles [51].

Correctness and Trust Model. Ensuring the correctness of StatusRoot and ActionRoot on the NSB is crucial. In § A.1, we specify a protocol that guarantees correctness for a permissioned NSB. In general, UIP is not limited by how the NSB is implemented. Practically, an alternative and lightweight of way of realizing the same functionality of the NSB is to implement the NSB as a multi-owner smart contract such that a threshold number (*e.g.,* the majority) of agreements from these owners are required to accept a certain StatusRoot or ActionRoot. This NSB construction also gives dApps and VESes more flexibility over which set of entities they deem trustworthy for adjudicating their operation status. This multi-owner contract should be deployed on the same BN as the ISC, so that the ISC can natively use the contract output for arbitration.

3 PROTOCOL DETAILS

In this section, we present the full protocol details. We start with the cryptography abstraction of the UIP in form of an *ideal functionality* $\mathcal{F}_{\text{UIP}}$. The ideal functionality defines the correctness and security properties that UIP wishes to attain by assuming a trusted entity. Then we present Prot_{UIP}, the decentralized real-world protocol that provably realizes $\mathcal{F}_{\text{UIP}}$, *i.e.,* Prot_{UIP} achieves the same functionality and security properties without assuming any trusted authorities.

3.1 Ideal Functionality $\mathcal{F}_{\text{UIP}}$

3.1.1 *Protocol Specification of* $\mathcal{F}_{\text{UIP}}$

$\mathcal{F}_{\text{UIP}}$ specifies the ideal functionality of a secure protocol that enables a pair of parties $(\mathcal{P}_a, \mathcal{P}_z)$ (*e.g.,* an dApp and a VES) to perform inter-BN operations. The detailed description of $\mathcal{F}_{\text{UIP}}$ is given in Figure 6. $\mathcal{F}_{\text{UIP}}$ defines the following interfaces.

Session Setup. Through this interface, a pair of parties $(\mathcal{P}_a, \mathcal{P}_z)$ requests $\mathcal{F}_{\text{UIP}}$ to securely realize an inter-BN operation Op. The Op is represented by a transaction intent graph $\mathcal{G}_T$ and an insurance contract *contract*. As a trusted entity, $\mathcal{F}_{\text{UIP}}$ generates keys for both parties, allowing $\mathcal{F}_{\text{UIP}}$ to sign transactions and compute certificates on their behalf. Both parties are required to stake sufficient funds, derived from the *contract*, into $\mathcal{F}_{\text{UIP}}$. $\mathcal{F}_{\text{UIP}}$ annotates each transaction intent $\mathcal{T}$ in $\mathcal{G}_T$ with its status (initialized to be unknown), its open/close timestamps (initialized to 0s), and its on-chain counterpart $\widetilde{T}$ (initialized to be empty). To accurately match $\mathcal{F}_{\text{UIP}}$ with the real-world protocol Prot_{UIP}, in Figure 6, we assume that $\mathcal{P}_a$ is the original requester of Op (*e.g.,* a dApp), and $\mathcal{P}_z$ is the execution driver of Op (*e.g.,* a VES).

Since $\mathcal{F}_{\text{UIP}}$ does not impose any special requirements on the underlying BNs, we model the ideal-world blockchain as an ideal functionality $\mathcal{F}_{\text{blockchain}}$ that supports two simple interfaces: (i) public ledger query and (ii) state transition triggered by transactions (where $\mathcal{F}_{\text{UIP}}$ imposes no constraint on both the ledger format and the consensus logic of state transitions).

Transaction Open/Close Requests. This pair of interfaces are called by either $\mathcal{P}_a$ or $\mathcal{P}_z$ to request opening and closing of transaction intents in $\mathcal{G}_T$. Upon receiving an opening request of $\mathcal{T}$, $\mathcal{F}_{\text{UIP}}$ checks $\mathcal{G}_T$ to decide whether $\mathcal{T}$ is eligible to be opened. If so, $\mathcal{F}_{\text{UIP}}$

```
Init: Data := $\emptyset$
Upon Receive SessionCreate($\mathcal{G}_T$, contract, $\mathcal{P}_a$, $\mathcal{P}_z$):
    generate the session ID sid $\leftarrow \{0,1\}^\lambda$ and keys for both parties
    send Cert([sid, $\mathcal{G}_T$, contract]; $\text{Sig}^{\mathcal{P}_z}_{\text{sid}}$, $\text{Sig}^{\mathcal{P}_a}_{\text{sid}}$) to both parties
    halt until both parties deposit sufficient fund, denoted as stake
    start a blockchain monitoring daemon for this session
    set an expiration timer timer for executing the contract term
    for $\mathcal{T} \in \mathcal{G}_T$ : initialize the annotations for $\mathcal{T}$
    update Data[sid] := {$\mathcal{G}_T$, contract, stake, timer}
Upon Receive ReqTransOpen($\mathcal{T}$, sid, $\mathcal{P}_z$):
    ($\mathcal{G}_T$, _, _, _) := Data[sid]; abort if not found
    assert $\mathcal{T}$ is eligible to be opened according to the state of $\mathcal{G}_T$
    compute $\widetilde{T}$ using the key of $\mathcal{T}$'s originator
    post $\widetilde{T}$ on $\mathcal{F}_{\text{blockchain}}$ for on-chain execution
    update $\mathcal{T}$.status := open, $\mathcal{T}$.$\text{ts}_{\text{open}}$ := now() and $\mathcal{T}$.trans := $\widetilde{T}$
    compute $\text{Atte}^o_{\mathcal{T}}$ := Cert([$\widetilde{T}$, open, sid, $\mathcal{T}$, $\text{ts}_{\text{open}}$]; $\text{Sig}^{\mathcal{P}_a}_{\text{sid}}$, $\text{Sig}^{\mathcal{P}_z}_{\text{sid}}$)
    send $\text{Atte}^o_{\mathcal{T}}$ to both parties to notify actions
Upon Receive ReqTransClose($\mathcal{T}$, sid, $\mathcal{P} \in (\mathcal{P}_a, \mathcal{P}_z)$):
    ($\mathcal{G}_T$, _, _, _) := Data[sid]; abort if not found
    abort if $\mathcal{T}$.trans is not instantiated (still an empty value)
    get $\widetilde{T}$ := $\mathcal{T}$.trans and query the ledger of $\mathcal{F}_{\text{blockchain}}$ for $\widetilde{T}$'s status
    abort if $\widetilde{T}$ is not finalized on $\mathcal{F}_{\text{blockchain}}$
    update $\mathcal{T}$.status := closed and $\mathcal{T}$.$\text{ts}_{\text{closed}}$ := now()
    compute $\text{Atte}^c_{\mathcal{T}}$ := Cert([$\widetilde{T}$, closed, sid, $\mathcal{T}$, $\text{ts}_{\text{closed}}$]; $\text{Sig}^{\mathcal{P}_a}_{\text{sid}}$, $\text{Sig}^{\mathcal{P}_z}_{\text{sid}}$)
    send $\text{Atte}^c_{\mathcal{T}}$ to both parties to notify actions
Upon Receive TermExecution(sid, $\mathcal{P} \in (\mathcal{P}_a, \mathcal{P}_z)$) public:
    ($\mathcal{G}_T$, contract, stake, timer) := Data[sid]; abort if not found
    abort if timer has not expired
    # The following is the arbitration logic specified by contract
    initialize a map resp to record which party to blame
    compute eligible transactions set $\mathcal{S}$ given current state of $\mathcal{G}_T$
    for $\mathcal{T} \in \mathcal{S}$ :
        if $\mathcal{T}$.status = unknown : update resp[$\mathcal{T}$] =: $\mathcal{P}_z$
        elif $\mathcal{T}$.status = open : update resp[$\mathcal{T}$] =: $\mathcal{T}$.from
        elif $\mathcal{T}$.status = closed and deadline constraint fails :
            update resp[$\mathcal{T}$] =: $\mathcal{T}$.from
    financially revert all closed transactions if resp is not empty
    allocate funds based on financial terms and resp (if not empty)
    return any remaining funds in stake to corresponding senders
    remove the internal bookkeeping of sid from Data
```

Figure 6: The ideal functionality $\mathcal{F}_{\text{UIP}}$.

computes and signs its on-chain counterpart $\widetilde{T}$, and posts $\widetilde{T}$ on its destination BN for execution. Additionally, $\mathcal{F}_{\text{UIP}}$ updates the status and opening timestamp annotations of $\mathcal{T}$. For a closing request of $\mathcal{T}$, $\mathcal{F}_{\text{UIP}}$ retrieves its on-chain transaction $\widetilde{T}$ (aborting if $\widetilde{T}$ is not found, meaning $\mathcal{T}$ has not been correctly opened), and queries $\widetilde{T}$'s status on the ledger of its destination BN. If $\widetilde{T}$ has been finalized, $\mathcal{F}_{\text{UIP}}$ annotates $\mathcal{T}$ as closed. In both interfaces, $\mathcal{F}_{\text{UIP}}$ computes an attestation for the corresponding status, and sends it to both parties to formally notify the actions taken by $\mathcal{F}_{\text{UIP}}$.

Financial Term Execution. Upon the expiration of *timer*, both parties can invoke the TermExecution interface to trigger the execution of *contract* according to the state of Op. If Op is correctly finished, *i.e.,* all transactions in $\mathcal{G}_T$ are correctly closed without violating precondition and deadline constraints, $\mathcal{F}_{\text{UIP}}$ pays the pre-defined service fee to $\mathcal{P}_z$ (the assumed driver of Op) from $\mathcal{P}_a$'s stake and returns any remaining funds to their senders. Otherwise, $\mathcal{F}_{\text{UIP}}$ financially reverts all closed transactions in $\mathcal{G}_T$ by sending their NBGs to the corresponding reversion addresses. Then $\mathcal{F}_{\text{UIP}}$ decides which party is responsible for the violation using the following arbitration logic: (i) if $\mathcal{G}_T$ has any uninitialized transaction that is eligible to be opened, $\mathcal{P}_z$ is responsible for not driving initialization; (ii) if an eligible transaction $\mathcal{T}$ remains opened or $\mathcal{T}$ is closed but with violated deadline constraint, the originator of $\mathcal{T}$ is responsible. Then $\mathcal{F}_{\text{UIP}}$ allocate funds to penalize the misbehaved parties, which could be either party or both parties.

Verbose Definition of $\mathcal{F}_{\text{UIP}}$. We *intentionally* define $\mathcal{F}_{\text{UIP}}$ verbosely (that is, sending many signed messages) in order to accurately match $\mathcal{F}_{\text{UIP}}$ to the real world protocol Prot_{UIP}. The verbosity is necessary to prove our security theorems in § A.3. For instance, in the SessionCreate interface, $\mathcal{F}_{\text{UIP}}$ certifies ($\mathcal{G}_T$, *contract*, sid) on behalf of both parties to simulate the result of a successful handshake between two parties in the real world. Another example is that the attestations generated in the ReqTransOpen and ReqTransClose interfaces are not essential to ensure the correctness of execution due to the assumed trustworthiness of $\mathcal{F}_{\text{UIP}}$. However, $\mathcal{F}_{\text{UIP}}$ still publishes attestations to emulate the *side effects* of Prot_{UIP} in the real world. Emulation is crucial to prove that $\mathcal{F}_{\text{UIP}}$ UC-realizes Prot_{UIP}, as shown in § A.3.

3.1.2 *Correctness and Security Properties of $\mathcal{F}_{\text{UIP}}$*

With the assumed trustworthiness, it is not hard to see that $\mathcal{F}_{\text{UIP}}$ offers the following correctness and security properties. First, after the pre-agreed timeout, Op either finishes correctly with all precondition and deadline rules satisfied, or Op fails and is financially reverted. Second, regardless of the stage at which Op fails, $\mathcal{F}_{\text{UIP}}$ holds the misbehaved parties accountable for the failure. Third, if $\mathcal{F}_{\text{blockchain}}$ is modeled with bounded transaction finality latency, Op is guaranteed to finish correctly if both parties are honest. If we use the alternative transaction closing definition discussed in § 2.2.2 (*i.e.,* $\mathcal{T}$ is considered to be closed as long as $\widetilde{T}$ is added to the pending transaction pool of $\mathcal{F}_{\text{blockchain}}$), the above finality constraint of $\mathcal{F}_{\text{blockchain}}$ can be relaxed. Finally, $\mathcal{F}_{\text{UIP}}$, by design, makes the *contract* public. This is because in the real world protocol Prot_{UIP}, the status of Op is public both on the ISC and the NSB. We acknowledge such limitation in this paper and leave the support for privacy-preserving inter-BN operations to future work.

3.2 Grounding Preliminary Protocols

Although $\mathcal{F}_{\text{UIP}}$ is conceptually clear due to the assumed trusted authority, fulfilling $\mathcal{F}_{\text{UIP}}$ through fully decentralized protocols is more involved. Thus, we divide the main protocol Prot_{UIP} into *five* preliminary protocols. In particular, Prot_{VES} and $\text{Prot}_{\text{dApp}}$ are the protocols implemented by VESes and dApps, respectively, to coordinate cross-BN executions. Prot_{ISC} is a smart-contract based arbitrator that performs authoritative yet trust-free arbitration of the execution status of Op and ensures financial atomicity of Op. Prot_{NSB} is the protocol realization of the NSB. Lastly, Prot_{UIP} includes Prot_{BC}, the protocol realization of the general purpose $\mathcal{F}_{\text{blockchain}}$. Prot_{UIP} claims no innovation in Prot_{BC} compared with the state-of-the-art.

```
Init: Data := $\emptyset$
Upon Receive SessionSetup($intent$, $term$, $\mathcal{D}$) public:
  generate the session ID sid $\leftarrow \{0,1\}^{\lambda}$
  propose the tx dependency graph $\mathcal{G}_T := \{\mathcal{T}\}$ that fulfills $intent$
  call [cid, $contract$] := $\mathsf{Prot}_{\mathsf{ISC}}$.CreateContract($term$, $\mathcal{G}_T$)
  send Cert([sid, $\mathcal{G}_T$, $contract$]; $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$) to $\mathcal{D}$ for approval
  halt until Cert([sid, $\mathcal{G}_T$, $contract$]; $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$) is received
  package $contract$ as a valid transaction $\widetilde{contract}$
  call $\mathsf{Prot}_{\mathsf{BC}}$.Exec($\widetilde{contract}$) or $\mathsf{Prot}_{\mathsf{NSB}}$.Exec($\widetilde{contract}$) for deployment
  halt until $\widetilde{contract}$ is initialized on $\mathsf{Prot}_{\mathsf{BC}}$ or $\mathsf{Prot}_{\mathsf{NSB}}$
  call $\mathsf{Prot}_{\mathsf{ISC}}$.StakeFund to stake the required funds in $\mathsf{Prot}_{\mathsf{ISC}}$
  halt until $\mathcal{D}$ has staked its required funds in $\mathsf{Prot}_{\mathsf{ISC}}$
  initialize Data[sid] := {$\mathcal{G}_T$, cid, $S_{\mathsf{Atte}}=\emptyset$, $S_{\mathsf{Merk}}=\emptyset$}
Daemon Watching(sid, {$\mathsf{Prot}_{\mathsf{BC}}$, ...}) private:
  ($\mathcal{G}_T$, _, $S_{\mathsf{Atte}}$, $S_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  for each $\mathcal{T} \in \mathcal{G}_T$ :
    continue if $\mathcal{T}$.status is not opened
    identify $\mathcal{T}$'s on-chain counterpart $\widetilde{T}$
    continue if $\mathsf{Prot}_{\mathsf{BC}}$.Status($\widetilde{T}$) is not committed
    get $\mathsf{ts}_{\mathsf{closed}}$ := $\mathsf{Prot}_{\mathsf{NSB}}$.BlockHeight()
    compute $C^{\mathcal{T}}_{\mathsf{closed}}$ := Cert([$\widetilde{T}$, closed, sid, $\mathcal{T}$, $\mathsf{ts}_{\mathsf{closed}}$], $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$)
    call $\mathsf{Prot}_{\mathsf{dApp}}$.CloseTrans($C^{\mathcal{T}}_{\mathsf{closed}}$) to request closing attestation
    call $\mathsf{Prot}_{\mathsf{BC}}$.MerkleProof($\widetilde{T}$) to obtain a finalization proof of $\widetilde{T}$
    denote the finalization proof as $\mathsf{Merk}^{c\_1}_{\mathcal{T}}$ (the first part of $\mathsf{Merk}^{c}_{\mathcal{T}}$)
    update $S_{\mathsf{Atte}}$.Add($C^{\mathcal{T}}_{\mathsf{closed}}$) and $S_{\mathsf{Merk}}$.Add($\mathsf{Merk}^{c\_1}_{\mathcal{T}}$)
Daemon Watching(sid, $\mathsf{Prot}_{\mathsf{NSB}}$) private:
  ($\mathcal{G}_T$, _, $S_{\mathsf{Atte}}$, $S_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  watch four types of attestations {$\mathsf{Atte}^{\mathsf{id}}$, $\mathsf{Atte}^{o}$, $\mathsf{Atte}^{\mathsf{od}}$, $\mathsf{Atte}^{c}$}
  process $fresh$ attestations via self-implemented interfaces (see below)
  # Retrieve alternative closing attestations if necessary.
  for each $\mathcal{T} \in \mathcal{G}_T$ :
    if $\mathcal{T}$.status = opened and $\mathsf{Merk}^{c\_1}_{\mathcal{T}} \in S_{\mathsf{Merk}}$ :
      retrieve the Merkle root $R$ of $\mathsf{Merk}^{c\_1}_{\mathcal{T}}$
      call $\mathsf{Prot}_{\mathsf{NSB}}$.MerkleProof($R$) to obtain an inclusion proof $\mathsf{Merk}^{c\_2}_{\mathcal{T}}$
      continue if $\mathsf{Merk}^{c\_2}_{\mathcal{T}}$ is not available yet on $\mathsf{Prot}_{\mathsf{NSB}}$
      compute the complete proof $\mathsf{Merk}^{c}_{\mathcal{T}}$ := [$\mathsf{Merk}^{c\_1}_{\mathcal{T}}$, $\mathsf{Merk}^{c\_2}_{\mathcal{T}}$]
      update $\mathcal{T}$.status := closed and $S_{\mathsf{Merk}}$.Add($\mathsf{Merk}^{c}_{\mathcal{T}}$)
  compute eligible transaction set $\mathcal{S}$ using the current state of $\mathcal{G}_T$
  for each $\mathcal{T} \in \mathcal{S}$:
    continue if $\mathcal{T}$.status is not unknown
    if $\mathcal{T}$.from = $\mathsf{Prot}_{\mathsf{dApp}}$:
      compute $\mathsf{Atte}^{i}_{\mathcal{T}}$ := Cert([$\mathcal{T}$, init, sid]; $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$)
      call $\mathsf{Prot}_{\mathsf{dApp}}$.InitTrans($\mathsf{Atte}^{i}_{\mathcal{T}}$) to request initialization
      call $\mathsf{Prot}_{\mathsf{NSB}}$.AddAction($\mathsf{Atte}^{i}_{\mathcal{T}}$) to prove $\mathsf{Atte}^{i}_{\mathcal{T}}$ is sent
      update $S_{\mathsf{Atte}}$.Add($\mathsf{Atte}^{i}_{\mathcal{T}}$) and $\mathcal{T}$.status := init
      non-blocking wait until $\mathsf{Prot}_{\mathsf{NSB}}$.MerkleProof($\mathsf{Atte}^{i}_{\mathcal{T}}$) rt. $\mathsf{Merk}^{i}_{\mathcal{T}}$
      update $S_{\mathsf{Merk}}$.Add($\mathsf{Merk}^{i}_{\mathcal{T}}$)
    else: call $self$.SInitedTrans(sid, $\mathcal{T}$)
Upon Receive SInitedTrans(sid, $\mathcal{T}$) private:                                   Northbound
  ($\mathcal{G}_T$, _, $S_{\mathsf{Atte}}$, $S_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  compute and sign the on-chain counterpart $\widetilde{T}$ for $\mathcal{T}$
  compute $\mathsf{Atte}^{\mathsf{id}}_{\mathcal{T}}$ := Cert([$\widetilde{T}$, inited, sid, $\mathcal{T}$]; $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$)
  call $\mathsf{Prot}_{\mathsf{dApp}}$.InitedTrans($\mathsf{Atte}^{\mathsf{id}}_{\mathcal{T}}$) to request opening of initialized $\mathcal{T}$
  call $\mathsf{Prot}_{\mathsf{NSB}}$.AddAction($\mathsf{Atte}^{\mathsf{id}}_{\mathcal{T}}$) to prove $\mathsf{Atte}^{\mathsf{id}}_{\mathcal{T}}$ is sent
  update $S_{\mathsf{Atte}}$.Add($\mathsf{Atte}^{\mathsf{id}}_{\mathcal{T}}$) and $\mathcal{T}$.status := inited
  non-blocking wait until $\mathsf{Prot}_{\mathsf{NSB}}$.MerkleProof($\mathsf{Atte}^{\mathsf{id}}_{\mathcal{T}}$) returns $\mathsf{Merk}^{\mathsf{id}}_{\mathcal{T}}$
  update $S_{\mathsf{Merk}}$.Add($\mathsf{Merk}^{\mathsf{id}}_{\mathcal{T}}$)
Upon Receive RInitedTrans($\mathsf{Atte}^{\mathsf{id}}_{\mathcal{T}}$) public:                      Southbound
  assert $\mathsf{Atte}^{\mathsf{id}}_{\mathcal{T}}$ has the valid form of Cert([$\widetilde{T}$, inited, sid, $\mathcal{T}$]; $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$)
  (_, _, $S_{\mathsf{Atte}}$, $S_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  abort if the $\mathsf{Atte}^{i}_{\mathcal{T}}$ corresponding to $\mathsf{Atte}^{\mathsf{id}}_{\mathcal{T}}$ is not in $S_{\mathsf{Atte}}$
  assert $\widetilde{T}$ is correctly associated with the intent $\mathcal{T}$
  get $\mathsf{ts}_{\mathsf{open}}$ := $\mathsf{Prot}_{\mathsf{NSB}}$.BlockHeight()
  compute $\mathsf{Atte}^{o}_{\mathcal{T}}$ := Cert([$\widetilde{T}$, open, sid, $\mathcal{T}$, $\mathsf{ts}_{\mathsf{open}}$]; $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$)
  call $\mathsf{Prot}_{\mathsf{dApp}}$.OpenTrans($\mathsf{Atte}^{o}_{\mathcal{T}}$) to request opening for $\mathcal{T}$
  call $\mathsf{Prot}_{\mathsf{NSB}}$.AddAction($\mathsf{Atte}^{o}_{\mathcal{T}}$) to prove $\mathsf{Atte}^{o}_{\mathcal{T}}$ is sent
  update $S_{\mathsf{Atte}}$.Add($\mathsf{Atte}^{o}_{\mathcal{T}}$) and $\mathcal{T}$.status := open
  non-blocking wait until $\mathsf{Prot}_{\mathsf{NSB}}$.MerkleProof($\mathsf{Atte}^{o}_{\mathcal{T}}$) returns $\mathsf{Merk}^{o}_{\mathcal{T}}$
  update $S_{\mathsf{Merk}}$.Add($\mathsf{Merk}^{o}_{\mathcal{T}}$)
Upon Receive OpenTrans($\mathsf{Atte}^{o}_{\mathcal{T}}$) public:                                Northbound
  assert $\mathsf{Atte}^{o}_{\mathcal{T}}$ has valid form of Cert([$\widetilde{T}$, open, sid, $\mathcal{T}$, $\mathsf{ts}_{\mathsf{open}}$]; $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$)
  (_, _, $S_{\mathsf{Atte}}$, $S_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  abort if the $\mathsf{Atte}^{\mathsf{id}}_{\mathcal{T}}$ corresponding to $\mathsf{Atte}^{o}_{\mathcal{T}}$ is not in $S_{\mathsf{Atte}}$
  assert $\mathsf{ts}_{\mathsf{open}}$ is within a bounded margin with $\mathsf{Prot}_{\mathsf{NSB}}$.BlockHeight()
  compute $\mathsf{Atte}^{\mathsf{od}}_{\mathcal{T}}$ := Cert([$\widetilde{T}$, open, sid, $\mathcal{T}$, $\mathsf{ts}_{\mathsf{open}}$]; $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$)
  call $\mathsf{Prot}_{\mathsf{BC}}$.Exec($\widetilde{T}$) to trigger on-chain execution
  call $\mathsf{Prot}_{\mathsf{dApp}}$.OpenedTrans($\mathsf{Atte}^{\mathsf{od}}_{\mathcal{T}}$) to acknowledge request
  call $\mathsf{Prot}_{\mathsf{NSB}}$.AddAction($\mathsf{Atte}^{\mathsf{od}}_{\mathcal{T}}$) to prove $\mathsf{Atte}^{\mathsf{od}}_{\mathcal{T}}$ is sent
  update $S_{\mathsf{Atte}}$.Add($\mathsf{Atte}^{\mathsf{od}}_{\mathcal{T}}$) and $\mathcal{T}$.status := opened
  non-blocking wait until $\mathsf{Prot}_{\mathsf{NSB}}$.MerkleProof($\mathsf{Atte}^{\mathsf{od}}_{\mathcal{T}}$) returns $\mathsf{Merk}^{\mathsf{od}}_{\mathcal{T}}$
  update $S_{\mathsf{Merk}}$.Add($\mathsf{Merk}^{\mathsf{od}}_{\mathcal{T}}$)
Upon Receive OpenedTrans($\mathsf{Atte}^{\mathsf{od}}_{\mathcal{T}}$) public:                             Southbound
  ast. $\mathsf{Atte}^{\mathsf{od}}_{\mathcal{T}}$ has valid form of Cert([$\widetilde{T}$, open, sid, $\mathcal{T}$, $\mathsf{ts}_{\mathsf{open}}$]; $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$)
  (_, _, $S_{\mathsf{Atte}}$, _) := Data[sid]; abort if not found
  abort if the $\mathsf{Atte}^{o}_{\mathcal{T}}$ corresponding to $\mathsf{Atte}^{\mathsf{od}}_{\mathcal{T}}$ is not in $S_{\mathsf{Atte}}$
  update $S_{\mathsf{Atte}}$.Add($\mathsf{Atte}^{\mathsf{od}}_{\mathcal{T}}$) and $\mathcal{T}$.status := opened
Upon Receive CloseTrans($C^{\mathcal{T}}_{\mathsf{closed}}$) public:                              Bidirectional
  assert $C^{\mathcal{T}}_{\mathsf{closed}}$ has valid form of Cert([$\widetilde{T}$, closed, sid, $\mathcal{T}$, $\mathsf{ts}_{\mathsf{closed}}$], $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$)
  assert $\widetilde{T}$ is finalized on its destination BN
  assert $\mathsf{ts}_{\mathsf{closed}}$ is within a bounded margin with $\mathsf{Prot}_{\mathsf{NSB}}$.BlockHeight()
  (_, _, $S_{\mathsf{Atte}}$, _) := Data[sid]; abort if not found
  compute $\mathsf{Atte}^{c}_{\mathcal{T}}$ := Cert([$\widetilde{T}$, closed, sid, $\mathcal{T}$, $\mathsf{ts}_{\mathsf{closed}}$], $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$)
  call $\mathsf{Prot}_{\mathsf{dApp}}$.ClosedTrans($\mathsf{Atte}^{c}_{\mathcal{T}}$) to acknowledged request
  update $S_{\mathsf{Atte}}$.Add($\mathsf{Atte}^{c}_{\mathcal{T}}$) and $\mathcal{T}$.status := closed
Upon Receive ClosedTrans($\mathsf{Atte}^{c}_{\mathcal{T}}$) public:                              Bidirectional
  ast. $\mathsf{Atte}^{c}_{\mathcal{T}}$ has valid form of Cert([$\widetilde{T}$, closed, sid, $\mathcal{T}$, $\mathsf{ts}_{\mathsf{closed}}$], $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$)
  (_, _, $S_{\mathsf{Atte}}$, _) := Data[sid]; abort if not found
  abort if Cert([$\widetilde{T}$, closed, sid, $\mathcal{T}$, $\mathsf{ts}_{\mathsf{closed}}$], $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$) is not in $S_{\mathsf{Atte}}$
  update $S_{\mathsf{Atte}}$.Add($\mathsf{Atte}^{c}_{\mathcal{T}}$) and $\mathcal{T}$.status := closed
Daemon Redeem(sid) private:
  # Invoke the insurance contract periodically
  ($\mathcal{G}_T$, cid, $S_{\mathsf{Atte}}$, $S_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  for each $unclaimed$ $\mathcal{T} \in \mathcal{G}_T$:
    get the $\mathsf{Atte}_{\mathcal{T}}$ from $S_{\mathsf{Atte}} \bigcup S_{\mathsf{Merk}}$ with the most advanced status
    call $\mathsf{Prot}_{\mathsf{ISC}}$.InsuranceClaim(cid, $\mathsf{Atte}_{\mathcal{T}}$) to claim insurance
```

Figure 7: Protocol description of of $\mathsf{Prot}_{\mathsf{VES}}$. Gray background denotes non-blocking operations triggered by status updates on $\mathsf{Prot}_{\mathsf{NSB}}$. Interfaces annotated with *northbound* and *southbound* process transactions originated from $\mathsf{Prot}_{\mathsf{VES}}$ and $\mathsf{Prot}_{\mathsf{dApp}}$, respectively. Interfaces annotated with *bidirectional* are shared by all transactions.

3.2.1 *Execution Protocol by VESes*

$\mathsf{Prot}_{\mathsf{VES}}$ specifies the inter-BN execution protocol implemented by VESes. § 2.2, provides a high-level explanation of $\mathsf{Prot}_{\mathsf{VES}}$, and its base version. In this section, we elaborate on the full protocol details. Protocol description for $\mathsf{Prot}_{\mathsf{VES}}$ is given in Figure 7.

SessionSetup. $\mathsf{Prot}_{\mathsf{VES}}$ opens a public interface to listen for session creation requests from dApps. Each request specifies an inter-BN operation *intent* and the financial *term* associated with the operation. This interface performs two major tasks: (i) to generate the transaction dependency graph $\mathcal{G}_T$ that fulfills the *intent* and deploy a contract to insure the execution of $\mathcal{G}_T$. Based on the financial *term* and $\mathcal{G}_T$, $\mathsf{Prot}_{\mathsf{VES}}$ interacts with the protocol $\mathsf{Prot}_{\mathsf{ISC}}$ to generate the insurance *contract*. Then $\mathsf{Prot}_{\mathsf{VES}}$ calls either $\mathsf{Prot}_{\mathsf{BC}}$ or $\mathsf{Prot}_{\mathsf{NSB}}$ to deploy the *contract*, depending on the preference of $\mathsf{Prot}_{\mathsf{VES}}$ and $\mathsf{Prot}_{\mathsf{dApp}}$. Once *contract* is deployed, $\mathsf{Prot}_{\mathsf{VES}}$ initializes its internal bookkeeping for the session. The two notations S_{Atte} and S_{Merk} represent two sets that store attestations and Merkle proofs, respectively. In the base protocol, the insurance contract only takes input as attestations, whereas in $\mathsf{Prot}_{\mathsf{UIP}}$, both attestations and Merkle proofs can be used as inputs to $\mathsf{Prot}_{\mathsf{ISC}}$.

Transaction (Intent) Status. In $\mathsf{Prot}_{\mathsf{UIP}}$, the types of transaction status are extended to {unknown, init, inited, open, opened, closed}, where a latter state is considered more *advanced* than a former one. The immediate benefit of finer-grained status definition is that if off-chain negotiations between two parties fail at any step, they can prove precisely, in form of attestations, the status of all transactions in $\mathcal{G}_T$, which will ultimately be used by $\mathsf{Prot}_{\mathsf{ISC}}$ for arbitration.

Watching Services. $\mathsf{Prot}_{\mathsf{VES}}$ internally creates two watching services to *proactively* read the status of all session-relevant BNs and the NSB. In a BN watching daemon, $\mathsf{Prot}_{\mathsf{VES}}$ reads the public ledger of $\mathsf{Prot}_{\mathsf{BC}}$. If $\mathsf{Prot}_{\mathsf{VES}}$ notices the finalization of an on-chain transaction $\widetilde{T}$ for any opened intent $\mathcal{T}$, it initiates the closing process for $\mathcal{T}$ by sending $\mathsf{Prot}_{\mathsf{dApp}}$ a timestamped certificate C_{closed}. Further, $\mathsf{Prot}_{\mathsf{VES}}$ retrieves a Merkle Proof from $\mathsf{Prot}_{\mathsf{BC}}$ to prove the finalization of $\widetilde{T}$. In § 2.2.4, a complete transaction closing proof consists of two individual Merkle proofs. We thus denote the finalization Merkle proof obtained from $\mathsf{Prot}_{\mathsf{BC}}$ as $\mathsf{Merk}_{\mathcal{T}}^{c\_1}$, representing the first part of the closing proof. $\mathsf{Merk}_{\mathcal{T}}^{c\_1}$ may also include a hash chain to prove the state resulting from $\widetilde{T}$ if $\mathsf{Prot}_{\mathsf{ISC}}$ later requires such information to verify the association between $\mathcal{T}$ and $\widetilde{T}$.

In the $\mathsf{Prot}_{\mathsf{NSB}}$ watching service, $\mathsf{Prot}_{\mathsf{VES}}$ performs following tasks. First, it searches the sorted Merkle trees rooted at ActionRoots for attestations that have not been received via the off-chain channel between $\mathsf{Prot}_{\mathsf{VES}}$ and $\mathsf{Prot}_{\mathsf{dApp}}$. This search enables the usage of $\mathsf{Prot}_{\mathsf{NSB}}$ as a publically-observable *fallback* communication medium for the off-chain channel. $\mathsf{Prot}_{\mathsf{VES}}$ watches for four types of fresh attestations $\{\mathsf{Atte}^{\mathsf{id}}, \mathsf{Atte}^{o}, \mathsf{Atte}^{\mathsf{od}}, \mathsf{Atte}^{c}\}$, and then calls the corresponding interfaces to handle them. Second, for each opened $\mathcal{T}$ whose closing attestation is still missing after $\mathsf{Prot}_{\mathsf{VES}}$ has sent C_{closed} (indicating slow or no reaction from $\mathsf{Prot}_{\mathsf{dApp}}$), $\mathsf{Prot}_{\mathsf{VES}}$ calls the MerkleProof interface of $\mathsf{Prot}_{\mathsf{NSB}}$ to check whether the second part of closing Merkle proof, denoted as $\mathsf{Merk}_{\mathcal{T}}^{c\_2}$, is available. If so, $\mathsf{Prot}_{\mathsf{VES}}$ concatenates $\mathsf{Merk}_{\mathcal{T}}^{c\_1}$ and $\mathsf{Merk}_{\mathcal{T}}^{c\_2}$ to construct a *complete* proof $\mathsf{Merk}_{\mathcal{T}}^{c}$, and then updates the status of $\mathcal{T}$ accordingly. Finally, based on the current state of $\mathcal{G}_T$, $\mathsf{Prot}_{\mathsf{VES}}$ computes a new set of transaction intents that are eligible to be opened. Then, $\mathsf{Prot}_{\mathsf{VES}}$ processes them by either requesting initialization from $\mathsf{Prot}_{\mathsf{dApp}}$ or calling SInitedTrans internally, depending on the originators of these transactions.

We now clarify how to create proof-of-actions (PoAs) using the $\mathsf{Prot}_{\mathsf{NSB}}$ protocol (an overview is given in § 2.2.4). While sending $\mathsf{Atte}_{\mathcal{T}}^{i}$ to $\mathsf{Prot}_{\mathsf{dApp}}$ over the off-chain channel, $\mathsf{Prot}_{\mathsf{VES}}$ further calls the AddAction interface of $\mathsf{Prot}_{\mathsf{NSB}}$ to permanently and publicly store $\mathsf{Atte}_{\mathcal{T}}^{i}$ on $\mathsf{Prot}_{\mathsf{NSB}}$. Later, in order to prove that it has computed and published $\mathsf{Atte}_{\mathcal{T}}^{i}$, $\mathsf{Prot}_{\mathsf{VES}}$ just needs to retrieve a Merkle proof $\mathsf{Merk}_{\mathcal{T}}^{i}$ from $\mathsf{Prot}_{\mathsf{NSB}}$, which essentially is a hash chain linking $\mathsf{Atte}_{\mathcal{T}}^{i}$ to an ActionRoot on a committed block of the NSB. This Merkle proof is referred to as a PoA. The Merkle proof retrieval is a non-blocking operation triggered by status update on $\mathsf{Prot}_{\mathsf{NSB}}$. Hereafter, we use $\mathsf{Merk}_{\mathcal{T}}^{i}(R)$ to refer to the root in $\mathsf{Merk}_{\mathcal{T}}^{i}$ (such notation applies for all other types of transaction status).

Northbound and Southbound Transactions. SInitedTrans and OpenTrans are two interfaces processing *northbound* transactions originating from $\mathsf{Prot}_{\mathsf{VES}}$. The SInitedTrans interface, invoked by the $\mathsf{Prot}_{\mathsf{NSB}}$ watching daemon, computes an inited attestation $\mathsf{Atte}_{\mathcal{T}}^{\mathsf{id}}$ to claim the initialization of $\mathcal{T}$, and then it sends $\mathsf{Atte}_{\mathcal{T}}^{\mathsf{id}}$ to $\mathsf{Prot}_{\mathsf{dApp}}$ for subsequent processing. OpenTrans listens for timestamped $\mathsf{Atte}_{\mathcal{T}}^{o}$ from $\mathsf{Prot}_{\mathsf{dApp}}$. Then it performs a correctness check against $\mathsf{Atte}_{\mathcal{T}}^{o}$, ensuring that the $\mathsf{Atte}_{\mathcal{T}}^{o}$ itself is valid, the immediately preceding attestation $\mathsf{Atte}_{\mathcal{T}}^{\mathsf{id}}$ exists, and the added $\mathsf{ts}_{\mathsf{open}}$ is genuine. If these checks pass, $\mathsf{Prot}_{\mathsf{VES}}$ dispatches $\widetilde{T}$ for on-chain execution, and uses $\mathsf{Prot}_{\mathsf{NSB}}$ to prove this action.

In OpenTrans, the difference between a pre-open attestation $\mathsf{Atte}_{\mathcal{T}}^{o}$ received from $\mathsf{Prot}_{\mathsf{dApp}}$ and a post-open (*i.e.*, opened) attestation $\mathsf{Atte}_{\mathcal{T}}^{\mathsf{od}}$ computed by $\mathsf{Prot}_{\mathsf{VES}}$ is that the opened attestations are signed by both parties. Only the $\mathsf{ts}_{\mathsf{open}}$ specified in $\mathsf{Atte}_{\mathcal{T}}^{\mathsf{od}}$ is used by $\mathsf{Prot}_{\mathsf{ISC}}$ to evaluate the deadline constraint of $\mathcal{T}$. When using $\mathsf{Merk}_{\mathcal{T}}^{c}$ as the closing attestation, $\mathsf{Prot}_{\mathsf{ISC}}$ uses the block number of $\mathsf{Merk}_{\mathcal{T}}^{c\_2}(R)$ as the $\mathsf{ts}_{\mathsf{closed}}$ (*c.f.,* § 3.2.3).

Southbound transactions originating from $\mathsf{Prot}_{\mathsf{dApp}}$ are handled similarly using the RInitedTrans and OpenedTrans interfaces.

CloseTrans and ClosedTrans. This pair of interfaces negotiate closing attestations. They can be used for both northbound and southbound transactions, depending on which party initiates the closing negotiation. In general, a transaction's originator has a stronger motivation to start the closing negotiation because the originator would be held accountable if the transaction were not timely closed by its deadline.

Insurance Claim. $\mathsf{Prot}_{\mathsf{VES}}$ periodically invokes $\mathsf{Prot}_{\mathsf{ISC}}$ to execute contract terms. All internally stored attestations and *complete* Merkle proofs are acceptable. However, for any $\mathcal{T}$ in $\mathcal{G}_T$, $\mathsf{Prot}_{\mathsf{VES}}$ should invoke $\mathsf{Prot}_{\mathsf{ISC}}$ only using the attestation or Merkle proof with the most advanced status, since lower-statused attestations for $\mathcal{T}$ are effectively ignored by $\mathsf{Prot}_{\mathsf{ISC}}$ (*c.f.,* § 3.2.3).

3.2.2 *Execution Protocol by dApps*

$\mathsf{Prot}_{\mathsf{dApp}}$ specifies the protocol implemented by dApps. $\mathsf{Prot}_{\mathsf{dApp}}$ defines the following set of interfaces to match $\mathsf{Prot}_{\mathsf{VES}}$. In particular, the InitedTrans and OpenedTrans match the SInitedTrans and

```
Init: Data := $\emptyset$
Upon Receive CreateContract($term$, $\mathcal{G}_T$):
  generate the contract code $contract$ as follows
  initialize three maps $T_{state}$, $A_{revs}$ and $F_{stake}$
  for each $\mathcal{T} \in \mathcal{G}_T$ :
    compute an internal identifier for $\mathcal{T}$ as tid := $H(\mathcal{T})$
    initialize $T_{state}$[tid] := [unknown, $\mathcal{T}$, $ts_{open}$=0, $ts_{closed}$=0, $st_{result}$]
    retrieve tid's fund-reversion account, denoted as dest, from $term$
    initialize $A_{revs}$[tid] := [amt=0, dest]
  compute an identifier for $contract$ as cid := $H(\overrightarrow{0}, contract)$
  initialize Data[cid] := [$\mathcal{G}_T$, $term$, $T_{state}$, $A_{revs}$, $F_{stake}$]
  send [cid, $contract$] to the requester for acknowledgment
Upon Receive StakeFund(cid):
  (_, _, _, _, $F_{stake}$) := Data[cid]; abort if not found
  update $F_{stake}$[msg.sender] := $F_{stake}$[msg.sender] + msg.value
Upon Receive InsuranceClaim(cid, Atte):
  (_, _, $T_{state}$, _, _) := Data[cid]; abort if not found
  compute tid := $H$(Atte.$\mathcal{T}$); T := $T_{state}$[tid] abort if not found
  abort if T.status is *more advanced* the status enclosed by Atte
  if Atte is a certificate signed by both parties :
    assert SigVerify(Atte) is true
    if Atte is $\mathsf{Atte}^{od}_{\mathcal{T}}$ : update T.status := opened; T.$ts_{open}$ := Atte.$ts_{open}$
    else : update T.status := closed; T.$ts_{closed}$ := Atte.$ts_{closed}$
  else : # Atte is in form of a Merkle proof
    assert MerkleVerify(Atte) is true
    if Atte is a $\mathsf{Merk}^{i}_{\mathcal{T}}$ or $\mathsf{Merk}^{id}_{\mathcal{T}}$ or $\mathsf{Merk}^{o}_{\mathcal{T}}$ :
      retrieve the certificate $\mathsf{Atte}^{i}_{\mathcal{T}}$ or $\mathsf{Atte}^{id}_{\mathcal{T}}$ or $\mathsf{Atte}^{o}_{\mathcal{T}}$ from Atte
      assert the $\widetilde{T}$ enclosed in $\mathsf{Atte}^{id}_{\mathcal{T}}$ or $\mathsf{Atte}^{o}_{\mathcal{T}}$ is genuine
      assert the $ts_{open}$ enclosed in $\mathsf{Atte}^{o}_{\mathcal{T}}$ is genuine
      update T.status := Atte.status
    elif Atte is $\mathsf{Merk}^{od}_{\mathcal{T}}$ :
      retrieve the certificate $\mathsf{Atte}^{od}_{\mathcal{T}}$ from Atte
      update T.status := opened and T.$ts_{open}$ := $\mathsf{Atte}^{od}_{\mathcal{T}}$.$ts_{open}$
    elif Atte is $\mathsf{Merk}^{c}_{\mathcal{T}}$ :
      update T.$st_{result}$ based on $\mathsf{Merk}^{c_{-1}}_{\mathcal{T}}$ if necessary
      update T.$ts_{closed}$ as the block height of $\mathsf{Merk}^{c_{-2}}_{\mathcal{T}}(R)$
      update T.status := closed
Upon Timeout SettleContract(cid):                                Internal Daemon
  ($\mathcal{G}_T$, $term$, $T_{state}$, $A_{revs}$, $F_{stake}$) := Data[cid]; abort if not found
  for (tid, T) $\in$ $T_{state}$ :
    continue if T.status is not closed
    update $A_{revs}$[tid].amt := T.$\mathcal{T}$.meta.amt
    if DeadlineVerify(T) = true : update T.status := correct
  compute $\mathcal{S}$ := DirtyTrans($\mathcal{G}_T$, $T_{state}$) # non-empty if Op fails.
  execute fund reversion for non-zero entries in $A_{revs}$ if $\mathcal{S}$ is not empty
  initialize a map $resp$ to record which party to blame
  for each (tid, T) $\in \mathcal{S}$ :
    if T.status = closed | open | opened : $resp$[tid] := T.$\mathcal{T}$.from
    elif T.status = inited : $resp$[tid] := T.$\mathcal{T}$.to
    elif T.status = init : $resp$[tid] := $\mathcal{D}$
    else : $resp$[tid] := $\mathcal{V}$
  execute additional terms (*e.g.,* penalty) in $term$ based on $resp$
  return any remaining funds in $F_{stake}$ to corresponding senders
  call Data.erase[cid] to stay silent afterwards
```

Figure 8: Trust-free Insurance Protocol $\mathsf{Prot}_{\mathsf{ISC}}$.

OpenTrans of $\mathsf{Prot}_{\mathsf{VES}}$, respectively, to process Atte^{id} and Atte^{od} sent by $\mathsf{Prot}_{\mathsf{VES}}$ when handling transactions originated from $\mathsf{Prot}_{\mathsf{VES}}$. The InitTrans and OpenTrans process Atte^{i} and Atte^{o} sent by $\mathsf{Prot}_{\mathsf{VES}}$ when executing transactions originated from $\mathsf{Prot}_{\mathsf{dApp}}$. The CloseTrans and ClosedTrans of $\mathsf{Prot}_{\mathsf{dApp}}$ match their counterparts in $\mathsf{Prot}_{\mathsf{VES}}$ to negotiate closing attestations. In § 3.3, we describe in detail the interaction between $\mathsf{Prot}_{\mathsf{dApp}}$ and $\mathsf{Prot}_{\mathsf{VES}}$.

For usability, UIP imposes smaller requirements on the watching daemons implemented by $\mathsf{Prot}_{\mathsf{dApp}}$. Specially, $\mathsf{Prot}_{\mathsf{dApp}}$ still proactively watches $\mathsf{Prot}_{\mathsf{NSB}}$ to have a fallback communication medium with $\mathsf{Prot}_{\mathsf{VES}}$ (watching for $\{\mathsf{Atte}^{i}, \mathsf{Atte}^{id}, \mathsf{Atte}^{o}, \mathsf{Atte}^{od}, \mathsf{Atte}^{c}\}$), and an alternative way of obtaining closing attestations as Merkle proofs. This requirement, however, is relaxed if $\mathsf{Prot}_{\mathsf{dApp}}$ chooses to use the base protocol defined in § 2.2 for certain op-intents. Further, $\mathsf{Prot}_{\mathsf{dApp}}$ is not required to proactively watch the status of underlying BNs or compute eligible transactions whenever the state of $\mathcal{G}_T$ changes. We intentionally offload such complexity on $\mathsf{Prot}_{\mathsf{VES}}$ to enable easy adoption of UIP by dApps. $\mathsf{Prot}_{\mathsf{dApp}}$, though, should (and is motivated to) check the status of self-originated transactions in order to initiate the negotiation for closure attestations.

3.2.3 Decentralized Insurance Contract $\mathsf{Prot}_{\mathsf{ISC}}$

$\mathsf{Prot}_{\mathsf{ISC}}$ defines the protocol of an authoritative yet trust-free arbitrator. § 2.2.3 gives an overview of $\mathsf{Prot}_{\mathsf{ISC}}$. The full protocol details of $\mathsf{Prot}_{\mathsf{ISC}}$ are given in Figure 8.

CreateContract. This interface is the entry point of creating insurance contract using $\mathsf{Prot}_{\mathsf{ISC}}$. It generates the contract code *contract* based on the given transaction dependency graph $\mathcal{G}_T$ and the financial *term*. The *contract* initializes useful internal bookkeeping data structures and opens three interfaces: StakeFund allows $\mathsf{Prot}_{\mathsf{VES}}$ and $\mathsf{Prot}_{\mathsf{dApp}}$ to stake funds into $\mathsf{Prot}_{\mathsf{ISC}}$, InsuranceClaim allows both parties to redeem attestations, and SettleContract executes the arbitration upon timeout. Compared with base protocol in § 2.2.3, the T_{state} for $\mathcal{T}$ may optionally store the resulting state st_{result} from $\widetilde{T}$ on $\widetilde{T}$'s destination BN if $\mathsf{Prot}_{\mathsf{ISC}}$ needs st_{result} to verify the association between $\mathcal{T}$ and $\widetilde{T}$ in the SettleContract interface.

InsuranceClaim. $\mathsf{Prot}_{\mathsf{ISC}}$ listens for attestations from both parties through this interface, based on which it updates internal state. The input of $\mathsf{Prot}_{\mathsf{ISC}}$ can be either in form of certificates signed by *both* parties (*i.e.,* Atte^{od} and Atte^{c}) or complete Merkle proofs. We explain several subtleties in the state update logic. First, when processing a Merkle proof $\mathsf{Merk}^{i}_{\mathcal{T}}$, $\mathsf{Merk}^{id}_{\mathcal{T}}$ or $\mathsf{Merk}^{o}_{\mathcal{T}}$, $\mathsf{Prot}_{\mathsf{ISC}}$ retrieves the single-party signed certificate $\mathsf{Atte}^{i}_{\mathcal{T}}$, $\mathsf{Atte}^{id}_{\mathcal{T}}$ or $\mathsf{Atte}^{o}_{\mathcal{T}}$ enclosed in the proof and performs the following correctness check against the certificate. (i) The certificate must be signed by the correct party, *i.e.,* $\mathsf{Atte}^{i}_{\mathcal{T}}$ is signed by $\mathcal{V}$, $\mathsf{Atte}^{id}_{\mathcal{T}}$ is signed by $\mathcal{T}$'s originator and $\mathsf{Atte}^{o}_{\mathcal{T}}$ is signed by the destination of $\mathcal{T}$. (ii) The enclosed on-chain transaction $\widetilde{T}$ in $\mathsf{Atte}^{id}_{\mathcal{T}}$ and $\mathsf{Atte}^{o}_{\mathcal{T}}$ is correctly associated with $\mathcal{T}$, where the precise definition of association is given in § 2.2.2. (iii)

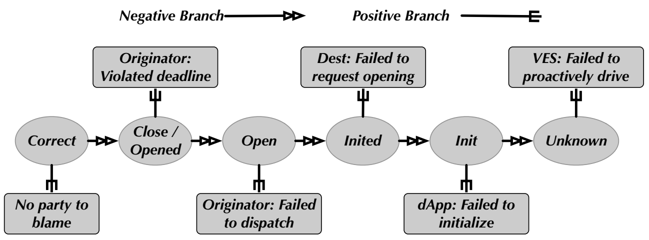

Negative Branch
Positive Branch
Originator: Violated deadline
Dest: Failed to request opening
VES: Failed to proactively drive
Correct
Close / Opened
Open
Inited
Init
Unknown
No party to blame
Originator: Failed to dispatch
dApp: Failed to initialize

Figure 9: The decision tree to decide the accountable party for a dirty transaction.

The enclosed ts_{open} in $\text{Atte}^{o}_{\mathcal{T}}$ is genuine, where the genuineness is defined as a bounded difference between ts_{open} and the block height of $\text{Merk}^{o}_{\mathcal{T}}(R)$. (iv) Only timestamps specified in opened and closed attestations, either in form of duel-signed certificates or Merkle proofs, are acceptable for updating T_{state}.

SettleContract. Prot_{ISC} registers a callback SettleContract to execute contract terms automatically upon timeout. Prot_{ISC} internally defines an additional transaction status, called correct. The status of a closed transaction is updated to correct if its deadline constraint is satisfied. Then, Prot_{ISC} computes the possible *dirty* transactions in $\mathcal{G}_T$, which are the transactions that are eligible to be opened, but with non-correct status. Thus, Op correctly finishes only if $\mathcal{G}_T$ has no dirty transactions.

If Op aborts prematurely, Prot_{ISC} employs a decision tree, shown in Figure 9, to unambiguously decide the responsible party for each dirty transaction. The decision tree is derived from the execution protocols Prot_{VES} and $\text{Prot}_{\text{dApp}}$. In particular, if a transaction $\mathcal{T}$'s status is closed, opened or open, then it is $\mathcal{T}$'s originator to blame for either failing to fulfill the deadline constraint or failing to dispatch $\widetilde{T}$ for on-chain execution. If a transaction $\mathcal{T}$'s status is inited, then it is $\mathcal{T}$'s destination party's responsibility for not proceeding with $\mathcal{T}$ even though $\text{Atte}^{\text{id}}_{\mathcal{T}}$ has been provably sent. If a transaction $\mathcal{T}$'s status is init (only transactions originated from dApp $\mathcal{D}$ can have init status), then $\mathcal{D}$ (the originator) is the party to blame for not reacting on the $\text{Atte}^{i}_{\mathcal{T}}$ sent by $\mathcal{V}$. Finally, if transaction $\mathcal{T}$'s status is unknown, then $\mathcal{V}$ is held accountable for not proactively driving the initialization of $\mathcal{T}$, no matter which party originates $\mathcal{T}$.

3.2.4 Prot_{BC} and Prot_{NSB}

Prot_{BC} specifies the protocol realization of a general-purpose blockchain where a set of miners run a secure consensus protocol to agree upon the public global state. In this paper, we regard Prot_{BC} as a conceptual party trusted for correctness and availability, *i.e.,* Prot_{BC} guarantees to correctly perform any predefined computation (*e.g.,* Turing-complete smart contract programs) and is always available to handle user requests despite unbounded response latency. Prot_{NSB} specifies the protocol realization of the network status blockchain discussed in § 2.2.4. Prot_{NSB} is a concrete instantiation of Prot_{BC} with several extensions, such as PoAs and status reporting of other BNs. Due to space constraint, we defer the detailed protocol description of Prot_{BC} and Prot_{NSB} in § A.2.

3.3 Protocol Description of Prot_{UIP}

As shown in Figure 10, our preliminary protocols enable concise yet informative description of Prot_{UIP}. In summary, Prot_{UIP} enables two parties, Prot_{VES} and $\text{Prot}_{\text{dApp}}$, to perform inter-BN operations,

```
                    **Op Initialization**
$\text{Prot}_{\text{dApp}}$: call $\text{Prot}_{\text{VES}}$.SessionSetup(*intent*, *term*)
$\text{Prot}_{\text{VES}}$: call $\text{Prot}_{\text{ISC}}$.CreateContract($\mathcal{G}_T$, *term*) to obtain *contract*
$\text{Prot}_{\text{VES}}$ & $\text{Prot}_{\text{dApp}}$: proceed if $\mathcal{G}_T$ and *contract* are mutually agreed
$\text{Prot}_{\text{VES}}$: call $\text{Prot}_{\text{NSB}}$.Exec($\widetilde{contract}$) or $\text{Prot}_{\text{BC}}$.Exec($\widetilde{contract}$)
$\text{Prot}_{\text{VES}}$ & $\text{Prot}_{\text{dApp}}$: proceed if $\widetilde{contract}$ is initiated and funded
                    **Op Execution**
$\text{Prot}_{\text{VES}}$: proactively watch $\text{Prot}_{\text{NSB}}$ and all {$\text{Prot}_{\text{BC}}$, ...} of Op
$\text{Prot}_{\text{dApp}}$: proactively watch $\text{Prot}_{\text{NSB}}$ (if necessary)
**while** $\mathcal{G}_T$ is not finished **and** not aborted **:**
 $\text{Prot}_{\text{VES}}$: computes the eligible transaction set $\mathcal{S}$ of $\mathcal{G}_T$
 process each transaction $\mathcal{T}$ of $\mathcal{S}$ in parallel
 **if** $\mathcal{T}$.from = $\text{Prot}_{\text{dApp}}$ **:**
   $\text{Prot}_{\text{VES}}$: $\text{Prot}_{\text{dApp}}$.InitTrans($\text{Atte}^{i}_{\mathcal{T}}$);$\text{Prot}_{\text{NSB}}$.AddAction($\text{Atte}^{i}_{\mathcal{T}}$)
   $\text{Prot}_{\text{dApp}}$: $\text{Prot}_{\text{VES}}$.RInitedTrans($\text{Atte}^{\text{id}}_{\mathcal{T}}$);$\text{Prot}_{\text{NSB}}$.AddAction($\text{Atte}^{\text{id}}_{\mathcal{T}}$)
   $\text{Prot}_{\text{VES}}$: $\text{Prot}_{\text{dApp}}$.OpenTrans($\text{Atte}^{o}_{\mathcal{T}}$);$\text{Prot}_{\text{NSB}}$.AddAction($\text{Atte}^{o}_{\mathcal{T}}$)
   $\text{Prot}_{\text{dApp}}$: $\text{Prot}_{\text{VES}}$.OpenedTrans($\text{Atte}^{\text{od}}_{\mathcal{T}}$);$\text{Prot}_{\text{NSB}}$.AddAction($\text{Atte}^{\text{od}}_{\mathcal{T}}$)
   $\text{Prot}_{\text{dApp}}$: $\text{Prot}_{\text{BC}}$.Exec($\widetilde{T}$)
   $\text{Prot}_{\text{VES}}$ & $\text{Prot}_{\text{dApp}}$: get $\text{Merk}^{c\_1}_{\mathcal{T}}$ from $\text{Prot}_{\text{BC}}$.MerkleProof($\widetilde{T}$)
   $\text{Prot}_{\text{dApp}}$: $\text{Prot}_{\text{VES}}$.CloseTrans($C^{\mathcal{T}}_{\text{closed}}$)
   $\text{Prot}_{\text{VES}}$: $\text{Prot}_{\text{dApp}}$.ClosedTrans($\text{Atte}^{c}_{\mathcal{T}}$)
   $\text{Prot}_{\text{VES}}$ & $\text{Prot}_{\text{dApp}}$: get $\text{Merk}^{c\_2}_{\mathcal{T}}$ from $\text{Prot}_{\text{NSB}}$.MerkleProof($\widetilde{T}$)
 else : # $\mathcal{T}$.from = $\text{Prot}_{\text{VES}}$
   $\text{Prot}_{\text{VES}}$: $\text{Prot}_{\text{dApp}}$.InitedTrans($\text{Atte}^{\text{id}}_{\mathcal{T}}$);$\text{Prot}_{\text{NSB}}$.AddAction($\text{Atte}^{\text{id}}_{\mathcal{T}}$)
   $\text{Prot}_{\text{dApp}}$: $\text{Prot}_{\text{VES}}$.OpenTrans($\text{Atte}^{o}_{\mathcal{T}}$);$\text{Prot}_{\text{NSB}}$.AddAction($\text{Atte}^{o}_{\mathcal{T}}$)
   $\text{Prot}_{\text{VES}}$: $\text{Prot}_{\text{dApp}}$.OpenedTrans($\text{Atte}^{\text{od}}_{\mathcal{T}}$);$\text{Prot}_{\text{NSB}}$.AddAction($\text{Atte}^{\text{od}}_{\mathcal{T}}$)
   $\text{Prot}_{\text{VES}}$: $\text{Prot}_{\text{BC}}$.Exec($\widetilde{T}$)
   $\text{Prot}_{\text{VES}}$: get $\text{Merk}^{c\_1}_{\mathcal{T}}$ from $\text{Prot}_{\text{BC}}$.MerkleProof($\widetilde{T}$)
   $\text{Prot}_{\text{VES}}$: $\text{Prot}_{\text{dApp}}$.CloseTrans($C^{\mathcal{T}}_{\text{closed}}$)
   $\text{Prot}_{\text{dApp}}$: $\text{Prot}_{\text{VES}}$.ClosedTrans($\text{Atte}^{c}_{\mathcal{T}}$)
   $\text{Prot}_{\text{VES}}$: get $\text{Merk}^{c\_2}_{\mathcal{T}}$ from $\text{Prot}_{\text{NSB}}$.MerkleProof($\widetilde{T}$)
 $\text{Prot}_{\text{VES}}$ & $\text{Prot}_{\text{dApp}}$: abort if violation detected or timeout
                    **Insurance Claim**
   $\text{Prot}_{\text{VES}}$ & $\text{Prot}_{\text{dApp}}$: call $\text{Prot}_{\text{ISC}}$.InsuranceClaim(Atte $\bigcup$ Merk)
```

Figure 10: Detailed protocol description of Prot_{UIP}. Gray background denotes the operations executed in parallel for each transaction in $\mathcal{G}_T$.

with PoA assistance from Prot_{NSB} and execution insurance from Prot_{ISC}. Prot_{UIP}, in total, has three phases: Op initialization, Op execution and insurance claim. In Op initialization, $\text{Prot}_{\text{dApp}}$ requests to start an inter-BN operation with Prot_{VES} by specifying an op-intent and financial terms. They rely on Prot_{ISC} to generate the *contract* and then deploy it on Prot_{NSB} or a mutually agreed Prot_{BC}. The Op execution phase starts after the initialization succeeds. At any time point, all eligible transactions are executed concurrently by both parties, for instance, by using dedicated threads. Given any step between Prot_{VES} and $\text{Prot}_{\text{dApp}}$, the originator of this step calls the corresponding interface defined by the terminator of the step via off-chain encrypted channels. Additionally, the originator uses Prot_{NSB} to prove its actions taken for this step. Either party may decide to abort the execution if it experiences timeout or violation. Finally, in the last phase, both parties call Prot_{ISC} to claim their rewards using dual-signed attestations or complete Merkle proofs collected from Prot_{NSB} and Prot_{BC}.

	Token-exchange Oriented dApps			Smart-contract Oriented dApps		UIP-powered dApps
Examples	*Central Exchange*	*Multi-owner Contract*	*Golden Token*	*Smart Contract Data Feed*		Atomizer & HyperService
Trusted Entities	Exchange	Contract owners	Golden tokens	Third-parties	Secure enclave	The NSB realization
Risks	Single-point failure	Owner collusion	Token failure	Manipulation	Enclave vulnerability	Exchange fluctuations
Limitations	Limited token types	Not inter-BN	Token trust	Central trust	Protocol overhead	Protocol overhead

Table 1: UIP-powered dApps advances the state-of-the-art blockchain applications.

3.4 Security Theorem

In this section, we claim the main security theorem of UIP. The correctness of Theorem 3.1 guarantees that $\mathsf{Prot}_{\mathsf{UIP}}$ achieves same security properties as $\mathcal{F}_{\mathsf{UIP}}$. The rigorous proof requires non-trivial simulator construction within the UC framework [17]. We provide the proof in § A.3.

Theorem 3.1. *Assuming that the distributed consensus algorithms used by relevant BNs are provably secure, the hash function is pre-image resistant, and the digital signature is EU-CMA secure (*i.e., *existentially unforgeable under a chosen message attack), our decentralized protocol $\mathsf{Prot}_{\mathsf{UIP}}$ securely UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{UIP}}$ against a malicious adversary in the passive corruption model.*

4 IMPLEMENTATION

In this section, we showcase two categories of dApps built upon UIP to demonstrate that UIP advances the state-of-the-art blockchain applications. Table 1 summarizes the results. We build and deploy our dApps on a prototype platform with 4 independent testnets. More implementation details are available in our source code [12].

4.1 Atomizer

As listed in Table 1, existing token exchange applications include centralized brokers (*e.g.,* Coinbase [2]), smart-contract based single-BN exchange (*e.g.,* IDEX [6]), and few cross-BN protocols where users are required to convert their tokens to the "golden tokens" issued by these protocols (*e.g.,* Bancor [1]). Powered by UIP, Atomizer surpasses these state-of-the-art by realizing a fully decentralized (*a.k.a.* trust-free) cross-BN exchange, where users can Atomizer with assurance due to its guaranteed global atomicity.

At a high level, the requests sent to Atomizer can be categorized into two types: (i) pre-agreed multi-party exchange where all involved accounts in the request and the amount of balance change of each account is given, and (ii) cross-BN token conversions where users just specify their conversion intents such as giving x tokens on one BN to exchange tokens on other two BNs. In the first case, Atomizer should ensure that the value of net balance update among all addresses is almost zero. In the second case, it is the Atomizer's responsibility to match a set of user requests that are zero-sum.

For both use cases, Atomizer approaches an autonomously selected VES with a zero-sum op-intent (see the illustrative example in Figure 2(a)). Then the VES constructs a transaction dependency graph $\mathcal{G}_T$ as follows: (i) it creates relay accounts on the set of BNs involved in the Op; (ii) it builds a graph of transaction intents between two Atomizer accounts or one Atomizer account and one relay account, and meanwhile specifies the amount of token transfer in each transaction; (iii) it topology-sorts the graph based on Atomizer-specified priorities (*i.e.,* precondition requirements); (iv) it populates the meta of each transaction intent to ensure linkability between transaction intents and their on-chain counterparts. For the Atomizer, the linkability is straightforward because these on-chain transactions are computable *a priori*. Thus, the VES simply needs to add the fullhash of an on-chain transaction to the meta of its corresponding transaction intent in $\mathcal{G}_T$.

4.2 HyperService

Financial derivatives are among the most commonly cited smart contract applications. However, external data feed, *i.e.,* an *oracle*, is often required for financial instructions. Currently, dApps relying on oracles using either third-party providers (*e.g.,* Oraclize [7]), or trusted hardware enclaves [51]. The HyperService dApp opens the possibility of *using BNs themselves as oracles*. With the built-in decentralization and correctness guarantee of BNs, HyperService fully avoids trusted parties while delivering genuine data feed to smart contracts. Essentially, it opens the possibility of practical smart contracting across a set of heterogeneous BNs, allowing dApps to fully utilize the strength of each underlying BN.

As an example, we implemented a HyperService dApp containing two contracts: a cash-settled Option contract which defines an agreement for a user to buy an asset from the contract owner at a price provided by a Broker contract lived on another BN. In this case, the $\mathcal{G}_T$ computed by the VES is conceptually simple with only two transaction intents (*i.e.,* $\mathcal{G}_T = \{\mathcal{T}_1, \mathcal{T}_2\}$) where $\mathcal{T}_1$ obtains a price from the Broker contract, and $\mathcal{T}_2$ triggers the Option contract with the obtained price. Both transactions originate from the VES.

The non-trivial part, however, is to verify that the price fed to the Option contract in $\mathcal{T}_2$ is indeed returned by the Broker contract upon the invocation by $\mathcal{T}_1$ (precisely speaking, its on-chain counterpart $\widetilde{\mathcal{T}}_1$). This exemplifies the case where some on-chain transactions (*i.e.,* $\widetilde{\mathcal{T}}_2$) cannot be decided a priori when computing $\mathcal{G}_T$, and $\mathsf{Prot}_{\mathsf{ISC}}$ further needs the state resulted by $\mathcal{T}_1$ to correctly verify the association between $\mathcal{T}_2$ and $\widetilde{\mathcal{T}}_2$. As detailed in § 3.2, UIP addresses this challenge by generating attestations that can prove resulting state from transactions. In our implementation, these attestations are constructed using the Merkle proofs that certifying the storage state of the Broker contract.

5 CONCLUSION

In this paper, we presented UIP, the first protocol that offers generic, secure, and financially safe inter-blockchain capability. UIP is able to operate on any blockchains with public ledgers, providing dApps with strong security assurance and financial atomicity. UIP achieves these properties by three innovative primitives: VES, ISC and NSB. We prove the security of UIP using the UC-framework. Our implementation demonstrates the practicality of UIP, as well as its capability of advancing the state-of-the-art blockchain applications.

This work does not raise any ethical issues.

A APPENDIX

A.1 Protocol Specification of Prot_{NSB} and $\mathcal{F}_{\text{blockchain}}$

The detailed protocol description of Prot_{NSB} and Prot_{BC} is given in Figure 11. We model block generation and consensus in Prot_{NSB} (and Prot_{BC}) as a *discrete clock* that proceeds in *epochs*. The length of an epoch is not fixed to reflect possible variance of the underlying consensus process among peers. At the end of each epoch, a new block is packaged and added to the append-only public Ledger.

The block format of Prot_{NSB} is shown in Figure 5. Each block packages two special Merkle trees (*i.e.,* ActionMT and StatusMT) and other Merkle trees (*e.g.,* TxMT) that are common in both Prot_{NSB} and Prot_{BC}. StatusMT and ActionMT are lexicographically sorted to store the items in the StatusPool and ActionPool, respectively. Considering the size limit of one block, some items in these pools may not be included (*e.g.,* due to lower gas prices), and will be rolled over to the next epoch.

StatusPool is built using the CloseureWatching and CloseureClaim interfaces. In particular, the CloseureWatching proactively watches all underlying BNs to retrieve both the transaction Merkle roots and state roots packaged in recently finalized blocks. It then adds these items into *Tx Roots* and *State Roots* maintained for each BN. When using finality-based transaction closure definition, this interface alone is sufficient because all Merkle-proof-based attestations required by the ISC can be built using the *Tx Roots* and *State Roots*. To support the alternative closure definition, Prot_{NSB} further needs the CloseureClaim interface. The CloseureClaim directly listens for transactions claimed by UIP users (VESes and dApps) for closure, rather than watching the transaction pending pools of all BNs. This design avoids including transactions that are not generated by any UIP sessions into the StatusMT.

Figure 11 also specifies the protocol of an honest peer/miner in the NSB to ensure the correctness of both interfaces. By complying with the protocol, honest peers accept any received claim (*i.e.,* a Merkle root or transaction claim) only after receiving a quorum of approvals for the claim. The protocol provably ensures the correctness of both interfaces, given that the number of Byzantine nodes in the permissioned NSB is no greater than $\mathcal{K}$ [37]. In CloseureClaim interface, an honest peer also conveys each claimed transaction into the transaction pool of its corresponding BN to ensure that the claimed transaction is in fact added to the pool.

The ActionPool is constructed in a similar manner as the StatusPool. In Figure 11, an interface annotated with override is also implemented by Prot_{BC}, although the implementation detail may be different; for instance Prot_{BC} may have different consensus process than Prot_{NSB} in the DiscreteTimer interface.

A.2 Alternative Transaction Closure Definition

Protocols described in § 3.2 consider the case where the closure of a transaction intent means that its on-chain counterpart has reached some level of finality on its BN. This definition places a greater responsibility on the VES because it must be sure that any transactions sent to the pending transaction pools of underlying BNs will eventually reach that level of finality before timing out. However, a transaction may fail to reach finality for a number of reasons beyond the control of the VES, such as regulatory reasons (*i.e.,* in permissioned blockchain), 51% attacks [10], and mining vulnerability [23].

The alternative closure definition is therefore proposed so that a VES needs only to ensure that its transactions reach the transaction pending pools. The finality responsibility, instead, is shifted to the dApp. The rational behind this design is that since the dApp dictates the content of $\mathcal{G}_T$ directly or indirectly (recall that Prot_{VES} halts until $\text{Prot}_{\text{dApp}}$ approves $\mathcal{G}_T$), the dApp author is most responsible for transactions which timely reach the transaction pending pools but do not reach finality in bounded time; in other worlds, failure to reach finality is considered a bug in the dApp.

We clarify the required protocol changes in Prot_{VES}, $\text{Prot}_{\text{dApp}}$ and Prot_{ISC} to support the alternative closure definition. In Prot_{VES}, the transaction closing negotiation, which currently is triggered by the watching daemon to Prot_{BC}, should be moved to the watching daemon to Prot_{NSB}. Meanwhile, the process of computing new eligible transactions, currently living in the watching service to Prot_{NSB}, should be moved to the watching service to Prot_{BC}. This is because regardless of the closure definition, a transactions intent can be opened only if all its preconditions are finalized. The $\text{Prot}_{\text{dApp}}$ is now completely relieved from watching any Prot_{BC} even for self-originated transactions since the closure negotiation will be triggered by status update on Prot_{NSB}.

The change required in Prot_{ISC} is more involved since transaction closure is no longer equivalent to finalization. As such, we need an additional transaction status *committed* such that Prot_{ISC} considers a transaction dirty if (i) it is closed with deadline rule violated or (ii) it is closed correctly but missing the attestation proving that it is committed. Prot_{ISC} blames the transaction originator only in the first dirty case, but blaming $\text{Prot}_{\text{dApp}}$ in the second case. Meanwhile, Prot_{VES} and $\text{Prot}_{\text{dApp}}$ need to add the corresponding negotiation procedures for attestations with committed status.

The final caveat in Prot_{ISC} is executing fund reversion for aborted Ops. At the time when Prot_{ISC} settles contract terms, it is difficult to decide whether Prot_{ISC} should financially revert a closed-but-uncommitted transaction; because if Prot_{ISC} does so but the transaction eventually gets removed from the pool, Prot_{ISC}'s arbitration is a "false positive", and vice versa. Since dApps are now held accountable for ensuring that transactions in $\mathcal{G}_T$ are committable, a reasonable choice is that Prot_{ISC} always financially reverts closed-but-unfinalized transactions out of the funds staked by $\text{Prot}_{\text{dApp}}$.

A.3 Formal Proof of Our Main Theorem

In this section, we prove our main Theorem 3.1. In the UC framework [17], the model of Prot_{UIP} execution is defined as a system of machines $(\mathcal{E}, \mathcal{A}, \pi_1, ..., \pi_n)$ where $\mathcal{E}$ is called the *environment*, $\mathcal{A}$ is the (real-world) adversary, and $(\pi_1, ..., \pi_n)$ are participants (referred to as *parties*) of Prot_{UIP} where each party may execute different parts of Prot_{UIP}. Intuitively, the environment $\mathcal{E}$ represents the *external* system that contains other protocols, including ones that provide inputs to, and obtain outputs from, Prot_{UIP}. The adversary $\mathcal{A}$ represents adversarial activity against the protocol execution, such as controlling communication channels and sending *corruption* messages to parties. $\mathcal{E}$ and $\mathcal{A}$ can communicate freely. The

```
Init: Data := ∅; Epoch := 0; Ledger := []
Daemon DiscreteTimer() override:
  continue if the current Epoch is not expired
  (TxPool, ActionPool, StatusPool) := Data[Epoch]
  initialize a block $\mathcal{B}$ with the format shown in Figure 5
  for pool ∈ (TxPool, ActionPool, StatusPool) :
    construct a (sorted) Merkle tree with selected items in pool
    populate $\mathcal{B}$ with the Merkle tree (TxMT, ActionMT, or StatusMT)
    remove these selected items from pool
  update Ledger.append($\mathcal{B}$) and execute trans. captured under TxMT
  start a new epoch Epoch := Epoch + 1
  initialize Data[Epoch] := [TxPool, ActionPool, StatusPool]
Daemon CloseureWatching({$\mathsf{Prot}_{\mathsf{BC}}$, ...}):
  proactively watch $\mathsf{Prot}_{\mathsf{BC}}$ for recently finalized blocks {$\mathcal{B}$, ...}
  (_, _, StatusPool) := Data[Epoch]; abort if not found
  retrieve the root $R_{tx}$ of TxMT and $R_{st}$ of StateMT on $\mathcal{B}$
  update StatusPool.Add($R_{tx}$, $R_{st}$)
  # Protocol of an honest peer $\mathcal{V}$ to ensure correctness
  Init: $\mathcal{V}$.StatusPool := []
  Daemon Watching({$\mathsf{Prot}_{\mathsf{BC}}$, ...}) and Watching($\mathcal{B}$, $\mathcal{S}$ = {Sig, ...})
    abort if $\mathcal{B}$ is not finalized on $\mathsf{Prot}_{\mathsf{BC}}$ or $\mathcal{B}$ is processed before
    abort if $\mathcal{S}$ contains more than $\mathcal{K}$ distinguished signatures
    retrieve the root $R_{tx}$ of TxMT and $R_{st}$ of StateMT on $\mathcal{B}$
    update $\mathcal{S}$.Add(Cert([$R_{tx}$, $R_{st}$]; $\mathsf{Sig}^{\mathcal{V}}$)); $\mathcal{V}$.StatusPool.Add($R_{tx}$, $R_{st}$)
    multicast ($\mathcal{B}$, $\mathcal{S}$) to other peers of the NSB
Daemon CloseureClaim($\widetilde{T}$):                    if Closure = PendingPool
  (_, _, StatusPool) := Data[Epoch]; abort if not found
  update StatusPool.Add($\widetilde{T}$)
  # Protocol of an honest peer $\mathcal{V}$ to ensure correctness
  Init: $\mathcal{V}$.StatusPool := []
  Daemon Watching($\widetilde{T}$, $\mathcal{S}$ = {Sig, ...})
    abort if $\widetilde{T}$ is already in $\mathcal{V}$.StatusPool
    abort if $\mathcal{S}$ contains more than $\mathcal{K}$ distinguished signatures
    add $\widetilde{T}$ to the pending transaction pool of its destination BN
    update $\mathcal{S}$.Add(Cert([$\widetilde{T}$]; $\mathsf{Sig}^{\mathcal{V}}$)); $\mathcal{V}$.StatusPool.Add($\widetilde{T}$)
    multicast ($\widetilde{T}$, $\mathcal{S}$) to other peers of the NSB
Upon Receive AddAction(Atte):
  (_, ActionPool, _) := Data[Epoch]; abort if not found
  update ActionPool.Add(Atte)
  # Similar correctness protocol as in CloseureClaim for honest peers
Upon Receive Exec($\widetilde{T}$) override:
  abort if $\widetilde{T}$ is not correctly constructed and signed
  (TxPool, _, _) := Data[Epoch]; abort if not found
  update TxPool.Add($\widetilde{T}$)
Upon Receive BlockHeight() override:
  return the block number of the last block on Ledger
Upon Receive MerkleProof(key) override:
  find the block $\mathcal{B}$ on Ledger containing key; abort if not found
  return a hash chain from key to the Merkle root on $\mathcal{B}$
```

Figure 11: Detailed protocol description of $\mathsf{Prot}_{\mathsf{NSB}}$. Interfaces annotated with override are also implemented by $\mathsf{Prot}_{\mathsf{BC}}$. Gray background denotes the protocol of honest peers in the NSB to ensure the correctness for the corresponding interface.

passive corruption model (used by Theorem 3.1) enables the adversary to observe the complete internal state of the corrupted party whereas the corrupted party is still protocol compliant, *i.e.,* the party executes instruction as desired. § A.3.4 discusses the *Byzantine* corruption model, where the adversary takes complete control of the corrupted party.

A.3.1 Proof Overview

To prove that $\mathsf{Prot}_{\mathsf{UIP}}$ UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{UIP}}$, we need to prove that $\mathsf{Prot}_{\mathsf{UIP}}$ *UC-emulates* $\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$, which is the *ideal protocol* (defined below) of our ideal functionality $\mathcal{F}_{\mathsf{UIP}}$. That is, for any adversary $\mathcal{A}$, there exists an adversary (often called simulator) $\mathcal{S}$ such that $\mathcal{E}$ cannot distinguish between the ideal world, featured by ($\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$, $\mathcal{S}$), and the real world, featured by ($\mathsf{Prot}_{\mathsf{UIP}}$, $\mathcal{A}$). Mathematically, on any input, the probability that $\mathcal{E}$ outputs $\vec{1}$ after interacting with ($\mathsf{Prot}_{\mathsf{UIP}}$, $\mathcal{A}$) in the real world differs by at most a negligible amount from the probability that $\mathcal{E}$ outputs $\vec{1}$ after interacting with ($\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$, $\mathcal{S}$) in the ideal world.

The ideal protocol $\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$ is a wrapper around $\mathcal{F}_{\mathsf{UIP}}$ by a set of dummy parties that have the same interfaces as the parties of $\mathsf{Prot}_{\mathsf{UIP}}$ in the real world. As a result, $\mathcal{E}$ is able to interact with $\mathcal{F}_{\mathsf{UIP}}$ in the ideal world the same way it interacts with $\mathsf{Prot}_{\mathsf{UIP}}$ in the real world. These dummy parties simply pass received input from $\mathcal{E}$ to $\mathcal{F}_{\mathsf{UIP}}$ and relay output of $\mathcal{F}_{\mathsf{UIP}}$ to $\mathcal{E}$, without implementing any additional logic. $\mathcal{F}_{\mathsf{UIP}}$ controls all keys of these dummy parties. For the sake of clear presentation, we abstract the real-world participants of $\mathsf{Prot}_{\mathsf{UIP}}$ as five parties {$\mathcal{P}_{\mathsf{VES}}$, $\mathcal{P}_{\mathsf{dApp}}$, $\mathcal{P}_{\mathsf{ISC}}$, $\mathcal{P}_{\mathsf{NSB}}$, $\mathcal{P}_{\mathsf{BC}}$ }. The corresponding dummy party of $\mathcal{P}_{\mathsf{VES}}$ in the ideal world is denoted as $\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$. This annotation applies for other parties.

Based on [17], to prove that $\mathsf{Prot}_{\mathsf{UIP}}$ UC-emulates $\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$ for any adversaries, it is sufficient to construct a simulator $\mathcal{S}$ just for the *dummy adversary* $\mathcal{A}$ that simply relays messages between $\mathcal{E}$ and the parties running $\mathsf{Prot}_{\mathsf{UIP}}$. The high-level process of the proof is that the simulator $\mathcal{S}$ observes the *side effects* of $\mathsf{Prot}_{\mathsf{UIP}}$ in the real world, such as attestation publication on the NSB and contract invocation of the ISC, and then accurately emulates these effects in the ideal world, with the help from $\mathcal{F}_{\mathsf{UIP}}$. As a result, $\mathcal{E}$ cannot distinguish the ideal and real worlds.

A.3.2 Construction of the Ideal Simulator $\mathcal{S}$

Next, we detail the construction of $\mathcal{S}$ by specifying what actions $\mathcal{S}$ should take upon observing instructions from $\mathcal{E}$. As a distinguisher, $\mathcal{E}$ sends the same instructions to the ideal world dummy parities as those sent to the real world parties.

- Upon $\mathcal{E}$ gives an instruction to start an inter-BN session between $\mathcal{P}^{\mathcal{I}}_{\mathsf{dApp}}$ and $\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$, $\mathcal{S}$ emulates the $\mathcal{G}_T$ and *contract* setup (*c.f.,* § A.3.3) and constructs a SessionCreate call to $\mathcal{F}_{\mathsf{UIP}}$ with parameter ($\mathcal{G}_T$, *contract*, $\mathcal{P}^{\mathcal{I}}_{\mathsf{dApp}}$, $\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$).
- To match the extended transaction status definition in $\mathsf{Prot}_{\mathsf{UIP}}$, additional interfaces (similar to the ReqTransOpen and ReqTransClose defined in Figure 6) are added in $\mathcal{F}_{\mathsf{UIP}}$, as shown in Figure 12. Note that there are slight updates in ReqTransOpen and ReqTransClose compared with the original definitions in Figure 6.

```
Upon Receive ReqTransInit($\mathcal{T}$, sid, $\mathcal{P}$):
    ($\mathcal{G}_T$, _, _, _) := Data[sid]; abort if not found
    assert $\mathcal{P}$ is $\mathcal{P}_z$
    assert $\mathcal{T}$ is eligible to be opened according the state of $\mathcal{G}_T$
    update $\mathcal{T}$.status := init
    compute $\mathsf{Atte}^i_{\mathcal{T}}$ := Cert([$\mathcal{T}$, init, sid]; $\mathsf{Sig}^{\mathcal{P}_z}_{\mathsf{sid}}$)
    send $\mathsf{Atte}^i_{\mathcal{T}}$ to both {$\mathcal{P}_a$, $\mathcal{P}_z$} to inform action
Upon Receive ReqTransInited($\mathcal{T}$, sid, $\widetilde{T}$, $\mathcal{P}$)
    ($\mathcal{G}_T$, _, _, _) := Data[sid]; abort if not found
    assert $\mathcal{P}$ = $\mathcal{T}$.from and $\mathcal{T}$.status = init
    compute the on-chain transaction $\widetilde{T}$ for $\mathcal{T}$
    update $\mathcal{T}$.status := inited and $\mathcal{T}$.trans := $\widetilde{T}$
    compute $\mathsf{Atte}^{id}_{\mathcal{T}}$ := Cert([$\widetilde{T}$, inited, sid, $\mathcal{T}$]; $\mathsf{Sig}^{\mathcal{P}}_{\mathsf{sid}}$)
    send $\mathsf{Atte}^{id}_{\mathcal{T}}$ to both {$\mathcal{P}_a$, $\mathcal{P}_z$} to inform action
Upon Receive ReqTransOpen($\mathcal{T}$, sid, $\widetilde{T}$, $\mathcal{P}$):
    ($\mathcal{G}_T$, _, _, _) := Data[sid]; abort if not found
    assert $\mathcal{P}$ = $\mathcal{T}$.to and $\mathcal{T}$.status = inited
    update $\mathcal{T}$.status = open and get $\mathsf{ts}_{\mathsf{open}}$ := now()
    compute $\mathsf{Atte}^{o}_{\mathcal{T}}$ := Cert([$\widetilde{T}$, open, $\mathsf{ts}_{\mathsf{open}}$, sid, $\mathcal{T}$]; $\mathsf{Sig}^{\mathcal{P}}_{\mathsf{sid}}$)
    send $\mathsf{Atte}^{o}_{\mathcal{T}}$ to both {$\mathcal{P}_a$, $\mathcal{P}_z$} to inform action
Upon Receive ReqTransOpened($\mathcal{T}$, sid, $\widetilde{T}$, $\mathcal{P}$, $\mathsf{ts}_{\mathsf{open}}$):
    ($\mathcal{G}_T$, _, _, _) := Data[sid]; abort if not found
    assert $\mathcal{P}$ = $\mathcal{T}$.from and $\mathcal{T}$.status = open
    assert $\mathsf{ts}_{\mathsf{open}}$ is within the error boundary with now()
    update $\mathcal{T}$.status = opened and get $\mathcal{T}.\mathsf{ts}_{\mathsf{open}}$ := $\mathsf{ts}_{\mathsf{open}}$
    post $\widetilde{T}$ on $\mathcal{F}_{\mathsf{blockchain}}$ for on-chain execution
    compute $\mathsf{Atte}^{od}_{\mathcal{T}}$ := Cert([$\widetilde{T}$, open, $\mathsf{ts}_{\mathsf{open}}$, sid, $\mathcal{T}$); $\mathsf{Sig}^{\mathcal{P}_a}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{P}_z}_{\mathsf{sid}}$)
    send $\mathsf{Atte}^{od}_{\mathcal{T}}$ to both {$\mathcal{P}_a$, $\mathcal{P}_z$} to inform action
Upon Receive ReqTransClose($\mathcal{T}$, sid, $\widetilde{T}$, $\mathsf{ts}_{\mathsf{closed}}$):
    ($\mathcal{G}_T$, _, _, _) := Data[sid]; abort if not found
    assert $\mathcal{T}$.status = opened
    get $\widetilde{T}$ := $\mathcal{T}$.trans and query the ledger of $\mathcal{F}_{\mathsf{blockchain}}$ for $\widetilde{T}$'s status
    abort if $T$ is not finalized on $\mathcal{F}_{\mathsf{blockchain}}$
    assert $\mathsf{ts}_{\mathsf{closed}}$ is within the error boundary with now()
    update $\mathcal{T}$.status := closed and $\mathcal{T}.\mathsf{ts}_{\mathsf{closed}}$ := $\mathsf{ts}_{\mathsf{closed}}$
    compute $\mathsf{Atte}^{c}_{\mathcal{T}}$ := Cert([$T$, closed, $\mathsf{ts}_{\mathsf{closed}}$, sid, $\mathcal{T}$); $\mathsf{Sig}^{\mathcal{P}_a}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{P}_z}_{\mathsf{sid}}$)
    send $\mathsf{Atte}^{c}_{\mathcal{T}}$ to both {$\mathcal{P}_a$, $\mathcal{P}_z$} to inform action
Upon Receive TermExecution(sid, $\mathcal{P} \in (\mathcal{P}_a, \mathcal{P}_z)$):
    # Similar to the SettleContract in Figure 8
```

Figure 12: Extension of the Ideal Functionality $\mathcal{F}_{\mathsf{UIP}}$.

- Upon $\mathcal{E}$ instructs $\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$ to send an initialization request for a transaction intent $\mathcal{T}$, $\mathcal{S}$ extracts $\mathcal{T}$ and sid from the instruction of $\mathcal{E}$, and constructs a ReqTransInit call to $\mathcal{F}_{\mathsf{UIP}}$ with parameter ($\mathcal{T}$, sid, $\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$). Other instructions in the same category are handled similarly by $\mathcal{S}$. In particular, for instruction to SInitedTrans, $\mathcal{S}$ calls ReqTransInited of $\mathcal{F}_{\mathsf{UIP}}$; for instructions to RInitedTrans, $\mathcal{S}$ calls ReqTransOpen of $\mathcal{F}_{\mathsf{UIP}}$; for instructions to OpenTrans, $\mathcal{S}$ calls ReqTransOpened of $\mathcal{F}_{\mathsf{UIP}}$; for instructions to CloseTrans, $\mathcal{S}$ calls ReqTransClose of $\mathcal{F}_{\mathsf{UIP}}$. $\mathcal{S}$ ignores instructions to OpenedTrans and ClosedTrans. $\mathcal{S}$ may also extract the $\widetilde{T}$ from the instruction, which is used by some interfaces of $\mathcal{F}_{\mathsf{UIP}}$ to verify the association between $\mathcal{T}$ and $\widetilde{T}$ (omitted in Figure 12).
- Due to the asymmetry of interfaces defined by $\mathcal{P}^{\mathcal{I}}_{\mathsf{dApp}}$ and $\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$, $\mathcal{S}$ acts slightly differently when observing instructions sent to $\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$. In particular, for instructions to InitTrans, $\mathcal{S}$ calls ReqTransInited of $\mathcal{F}_{\mathsf{UIP}}$; for instructions to InitedTrans, $\mathcal{S}$ calls ReqTransOpen of $\mathcal{F}_{\mathsf{UIP}}$; for instructions to OpenTrans, $\mathcal{S}$ calls ReqTransOpened of $\mathcal{F}_{\mathsf{UIP}}$. The rest handlings are the same as those of $\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$.
- Upon $\mathcal{E}$ instructs $\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$ to invoke the smart contract, $\mathcal{S}$ locally executes the *contract* and the instructs $\mathcal{F}_{\mathsf{UIP}}$ to published the updated *contract* to $\mathcal{P}^{\mathcal{I}}_{\mathsf{ISC}}$.

A.3.3 *Indistinguishability of Real and Ideal Worlds*

To prove indistinguishability of the real and ideal worlds from the perspective of $\mathcal{E}$, we will go through a sequence of *hybrid arguments*, where each argument is a hybrid construction of $\mathcal{F}_{\mathsf{UIP}}$, a subset of dummy parties of $\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$, and a subset of real-world parties of $\mathsf{Prot}_{\mathsf{UIP}}$, except that the first argument that is $\mathsf{Prot}_{\mathsf{UIP}}$ without any ideal parties and the last argument is $\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$ without any real world parties. We prove that $\mathcal{E}$ cannot distinguish any two consecutive hybrid arguments. Then based on the transitivity of protocol emulation [17], we prove that the first argument (*i.e.,* $\mathsf{Prot}_{\mathsf{UIP}}$) UC-emulates the last argument (*i.e.,* $\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$).

Real World. We start with the real world $\mathsf{Prot}_{\mathsf{UIP}}$ with a dummy adversary that simply passes messages to and from $\mathcal{E}$.

Hybrid A$_1$. Hybrid A_1 is the same as the real world, except that the ($\mathcal{P}_{\mathsf{VES}}$, $\mathcal{P}_{\mathsf{dApp}}$) pair is replaced by the dummy ($\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$, $\mathcal{P}^{\mathcal{I}}_{\mathsf{dApp}}$) pair. Upon observing an instruction from $\mathcal{E}$ to create an inter-BN session between the two parties with an op-intent *intent* and financial terms *term*, $\mathcal{S}$ first comes up with the transaction graph $\mathcal{G}_T$ that fulfills *intent*. Then $\mathcal{S}$ calls the CreateContract interface of $\mathcal{P}_{\mathsf{ISC}}$ (living in the Hybrid A_1) with parameter ($\mathcal{G}_T$, *term*) to obtain the contract code *contract*. Upon *contract* is received, $\mathcal{S}$ calls the SessionCreate interface of $\mathcal{F}_{\mathsf{UIP}}$ with parameter ($\mathcal{G}_T$, *contract*, $\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$, $\mathcal{P}^{\mathcal{I}}_{\mathsf{dApp}}$), which will output a certificate to both dummy parties to emulate the handshake result between $\mathcal{P}_{\mathsf{VES}}$ and $\mathcal{P}_{\mathsf{dApp}}$ in the real world. $\mathcal{S}$ also deploys *contract* on $\mathcal{P}_{\mathsf{NSB}}$ or $\mathcal{P}_{\mathsf{BC}}$ in the Hybrid A_1. Finally, $\mathcal{S}$ stakes required funds into $\mathcal{F}_{\mathsf{UIP}}$ to unblock its execution.

Upon observing an instruction from $\mathcal{E}$ (sent to either dummy parties) to execute a transaction in $\mathcal{G}_T$, based on its construction in § A.3.2, $\mathcal{S}$ has enough information to construct a call to $\mathcal{F}_{\mathsf{UIP}}$ with a proper interface and parameters. If the call generates an attestation Atte, $\mathcal{S}$ retrieves Atte to emulate the PoAs staking in the real world. In particular, if in the real world, $\mathcal{P}_{\mathsf{VES}}$ (and $\mathcal{P}_{\mathsf{dApp}}$) publishes an attestation on $\mathcal{P}_{\mathsf{NSB}}$ after receiving the same instruction from $\mathcal{E}$, then $\mathcal{S}$ publishes the corresponding attestation to $\mathcal{P}_{\mathsf{NSB}}$ in the Hybrid A_1 as well. Otherwise, $\mathcal{S}$ skip the publishing. Later, $\mathcal{S}$ retrieves (and stores) the Merkle proof from $\mathcal{P}_{\mathsf{NSB}}$ for a published attestation Atte, and then instructs $\mathcal{F}_{\mathsf{UIP}}$ to output the proof to the dummy party which is supposed (by $\mathcal{E}$) to be the publisher of Atte.

Upon observing an instruction from $\mathcal{E}$ (to either dummy party) to invoke the smart contract, $\mathcal{S}$ uses its saved attestations or Merkle proofs to invoke $\mathcal{P}_{\mathsf{ISC}}$ in the Hybrid A_1 accordingly.

Note that in the real world, the execution of Op is automatic in the sense that Op can proceed even without additional instructions from $\mathcal{E}$ after successful session setup. In the Hybrid A_1, although $\mathcal{P}_{\mathsf{VES}}$ and $\mathcal{P}_{\mathsf{dApp}}$ are replaced by dummy parties, $\mathcal{S}$, with fully knowledge of $\mathcal{G}_T$, is still able to drive the execution of Op so that from $\mathcal{E}$'s perspective, Op is executed automatically. Further, since $\mathcal{P}_{\mathsf{ISC}}$ still lives in the Hybrid A_1, $\mathcal{S}$ should not trigger the

TermExecution interface of $\mathcal{F}_{\mathsf{UIP}}$ to avoid double execution on the same contract terms. $\mathcal{S}$ can still reclaim its funds staked in $\mathcal{F}_{\mathsf{UIP}}$ via "backdoor" channels since $\mathcal{S}$ and $\mathcal{F}_{\mathsf{UIP}}$ are allowed to communicate freely under the UC framework.

Fact 1. *With the aforementioned construction of $\mathcal{S}$ and $\mathcal{F}_{\mathsf{UIP}}$, it is immediately clear that the outputs of both dummy parties in the Hybrid A_1 are exactly the same as the outputs of the corresponding actual parties in the real world, and all side effects in the real world are accurately emulated by $\mathcal{S}$ in the Hybrid A_1. Thus, $\mathcal{E}$ cannot distinguish with the real world and the Hybrid A_1.*

Hybrid A$_2$. Hybrid A_2 is the same as the Hybrid A_1, expect that $\mathcal{P}_{\mathsf{ISC}}$ is further replaced by the dummy $\mathcal{P}^{\mathcal{I}}_{\mathsf{ISC}}$. As a result, $\mathcal{S}$ is required to resume the responsibility of $\mathcal{P}_{\mathsf{ISC}}$ in the Hybrid A_2. In particular, when observing an instruction to initiate an Op, $\mathcal{S}$ computes the contract code *contract* based on the op-intent and financial term, and then instructs $\mathcal{F}_{\mathsf{UIP}}$ to publish the *contract* on $\mathcal{P}^{\mathcal{I}}_{\mathsf{ISC}}$, which is observable by $\mathcal{E}$. For any instruction to invoke *contract*, $\mathcal{S}$ locally executes *contract* with the input and then publishes the updated *contract* to $\mathcal{P}^{\mathcal{I}}_{\mathsf{ISC}}$ via $\mathcal{F}_{\mathsf{UIP}}$. Finally, upon the predefined contract timeout, $\mathcal{S}$ calls the TermExecution interface of $\mathcal{F}_{\mathsf{UIP}}$ with parameter $(\mathsf{sid}, \mathcal{P}^{\mathcal{I}}_{\mathsf{VES}})$ or $(\mathsf{sid}, \mathcal{P}^{\mathcal{I}}_{\mathsf{dApp}})$ to execute the *contract*, which emulates the arbitration performed by $\mathcal{P}_{\mathsf{ISC}}$ in the Hybrid A_1.

It is immediately clear that with the help of $\mathcal{S}$ and $\mathcal{F}_{\mathsf{UIP}}$, the output of the dummy $\mathcal{P}^{\mathcal{I}}_{\mathsf{ISC}}$ and side effects in the Hybrid A_2 are exactly the same as those in the Hybrid A_1. Thus, $\mathcal{E}$ cannot distinguish these two worlds.

Hybrid A$_3$. Hybrid A_3 is the same as the Hybrid A_2, expect that $\mathcal{P}_{\mathsf{NSB}}$ is further replaced by the dummy $\mathcal{P}^{\mathcal{I}}_{\mathsf{NSB}}$. Since the structure of $\mathcal{P}_{\mathsf{NSB}}$ and messages sent to $\mathcal{P}_{\mathsf{NSB}}$ are public, simulating its functionality by $\mathcal{S}$ is trivial. Therefore, Hybrid A_3 is identically distributed as Hybrid A_2 from the view of $\mathcal{E}$.

Hybrid A$_4$, *i.e.,* the ideal world. Hybrid A_4 is the same as the Hybrid A_3, expect that $\mathcal{P}_{\mathsf{BC}}$ (the last real-world party) is further replaced by the dummy $\mathcal{P}^{\mathcal{I}}_{\mathsf{BC}}$. Thus, the Hybrid A_4 is essentially $\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$. Since the functionality of $\mathcal{P}_{\mathsf{BC}}$ is a strict subset of that of $\mathcal{P}_{\mathsf{NSB}}$, simulating $\mathcal{P}_{\mathsf{BC}}$ by $\mathcal{S}$ is straightforward. Therefore, $\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$ is indistinguishable with the Hybrid A_3 from $\mathcal{E}$'s perspective.

Then given the transitivity of protocol emulation, we show that $\mathsf{Prot}_{\mathsf{UIP}}$ UC-emulates $\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$, and therefore prove that $\mathsf{Prot}_{\mathsf{UIP}}$ UC-realizes $\mathcal{F}_{\mathsf{UIP}}$. Throughout the simulation, we maintain a key invariant: $\mathcal{S}$ and $\mathcal{F}_{\mathsf{UIP}}$ together can always accurately simulate the desired outputs and side effects on all (dummy and real) parties in all Hybrid worlds. Thus, from $\mathcal{E}$'s view, the indistinguishability between the real and ideal worlds naturally follows.

A.3.4 Byzantine Corruption Model

Theorem 3.1 considers the passive corruption model. In this section, we discuss the more general Byzantine corruption model for $\mathcal{P}_{\mathsf{VES}}$ and $\mathcal{P}_{\mathsf{dApp}}$ (by assumption of this paper, blockchains and smart contracts are trusted for correctness). Previously, we construct $\mathcal{S}$ and $\mathcal{F}_{\mathsf{UIP}}$ accurately to match the *desired* execution of $\mathsf{Prot}_{\mathsf{UIP}}$. However, if one party is Byzantinely corrupted, the party behaves arbitrarily. As a result, if $\mathcal{P}_{\mathsf{VES}}$ and $\mathcal{P}_{\mathsf{dApp}}$ *only* rely on encrypted off-chain channels for protocol communication without publishing any actions on $\mathcal{P}_{\mathsf{NSB}}$, $\mathcal{S}$ is unable to accurately simulate such actions, resulting in possible difference between the ideal world and the real world from $\mathcal{E}$'s view.

To incorporate the Byzantine corruption model into our security analysis, we consider a variant of $\mathsf{Prot}_{\mathsf{UIP}}$, referred to as $\mathsf{H\text{-}Prot}_{\mathsf{UIP}}$, that requires $\mathcal{P}_{\mathsf{VES}}$ and $\mathcal{P}_{\mathsf{dApp}}$ to *only use* $\mathcal{P}_{\mathsf{NSB}}$ as the communication medium. Thus, all protocol execution steps taken by both parties are publicly observable by $\mathcal{S}$, allowing $\mathcal{S}$ to emulate whatever actions a (corrupted) part may take in the real world. Therefore, it is not hard to reach the following Lemma.

Lemma A.1. *With the same assumption of Theorem 3.1, the UIP protocol variant $\mathsf{H\text{-}Prot}_{\mathsf{UIP}}$ securely UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{UIP}}$ against a malicious adversary in the Byzantine corruption model.*

A.4 Architecture of VESes

In this section, we discuss the architecture of VESes, including VES discovery and the anatomy of a VES. In order to find a VES, a dApp relies on a *directory* (similar to the Tor's relay directories [9, 20]), which we envision to be an informal list of VESes together with their operation models. A dApp chooses a VES that matches their inter-BN operation requirements, including sufficient BN reachability by the VES, reasonable service fee, and sufficient trustworthiness for execution. For example, a dApp may prefer a VES with Intel SGX [19] capability for privacy-preserving computation [21, 46, 52]. Since all inter-BN execution results are publicly verifiable, it is possible to build a VES reputation system to provide a valuable metric for VES selection. The VES thus misbehaves at its own risk.

A VES itself is a distributed system, executing transactions across potentially several *isolation domains.* For example, the VES may execute lower-value transactions together, perhaps even on cloud infrastructure, but execute higher-value transactions on dedicated machines, limiting the risk of key exfiltration from operating system or even hardware vulnerabilities such as Meltdown, Spectre, and RowHammer [30, 31, 38]. If an isolation domain fails, the VES's risk is limited to Ops that are currently executing in that domain. Within an isolation domain, a VES may further operate several machines to limit the number of BNs in which each machine needs to participate as a fullnode, as well as balancing the mapping between transaction intents and nodes.

Furthermore, in order to improve availability, a VES may designate one or more backup nodes for each transaction intent. To ensure that each transaction is executed exactly once, the VES must ensure that no matter which node generates the transaction, the transaction is bitwise identical. Towards this end, the VES uses a common pool of randomness to generate random numbers (*e.g.,* keys) for that transaction. For example, the VES may assign each Op a secret randomness key k. Then for generating the i-th transaction, VES nodes use $\mathrm{PRF}_k(i)$ as a randomness seed; within the a transaction, to generate the j-th random number for transaction i, VES nodes use the value $\mathrm{PRF}_{\mathrm{PRF}_k(i)}(j)$.

Due to limitations in the size of funds that a VES may wish to stake to the ISC, a dApp may be too large for any single VES. Alternatively, a dApp may span a set of BNs such that no single VES has the reachability to all of them. In such cases, a dApp could be executed by a collection of VESes. In general, UIP is not limited by such multi-VES execution as long as the ISC is instantiated properly with correct reversion accounts and finer-grained arbitration logic.

REFERENCES

[1] Bancor. https://www.bancor.network.

[2] Coinbase. https://www.coinbase.com/.

[3] CoinMarketCap. https://coinmarketcap.com.

[4] Cosmos. https://cosmos.network.

[5] DPOS Consensus Algorithm. https://steemit.com/dpos/@dantheman/dpos-consensus-algorithm-this-missing-white-paper.

[6] IDEX. https://idex.market/.

[7] Oraclize. http://www.oraclize.it.

[8] Polkadot. https://polkadot.network.

[9] Tor Directory Authorities. https://metrics.torproject.org/rs.html#search/flag:authority.

[10] Ethereum Classic Might Have Been Hit by a 51% Attack. https://www.ccn.com/ethereum-classic-might-have-been-hit-by-a-51-attack/, 2019.

[11] Monoxide: Scale Out Blockchain with Asynchronized Consensus Zones. In *USENIX Symposium on Networked Systems Design and Implementation* (2019).

[12] Prototype of Universal Inter-Blockchain Protocol (UIP). https://sites.google.com/view/universal-inter-bn-protocol, 2019.

[13] Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., and Danezis, G. Chainspace: A Sharded Smart Contracts Platform. *Network and Distributed System Security Symposium* (2017).

[14] Breidenbach, L., Cornell Tech, I., Daian, P., Tramer, F., and Juels, A. Enter the Hydra: Towards Principled Bug Bounties and Exploit-Resistant Smart Contracts. In *27th USENIX Security Symposium* (2018).

[15] Buterin, V., et al. A Next-Generation Smart Contract and Decentralized Application Platform. *white paper* (2014).

[16] Caesar, M., and Rexford, J. BGP Routing Policies in ISP Networks. *IEEE network* (2005).

[17] Canetti, R. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *IEEE Symposium on Foundations of Computer Science* (2001).

[18] Cheng, R., Zhang, F., Kos, J., He, W., Hynes, N., Johnson, N., Juels, A., Miller, A., and Song, D. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. *arXiv preprint arXiv:1804.05141* (2018).

[19] Costan, V., and Devadas, S. Intel SGX explained. https://eprint.iacr.org/2016/086.pdf.

[20] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium* (2004).

[21] Dinh, T. T. A., Saxena, P., Chang, E.-C., Ooi, B. C., and Zhang, C. M2R: Enabling Stronger Privacy in MapReduce Computation. In *USENIX Security Symposium* (2015), pp. 447–462.

[22] Eyal, I., Gencer, A. E., Sirer, E. G., and Van Renesse, R. Bitcoin-NG: A Scalable Blockchain Protocol. In *USENIX Symposium on Networked Systems Design and Implementation* (2016), pp. 45–59.

[23] Eyal, I., and Sirer, E. G. Majority is not Enough: Bitcoin Mining is Vulnerable. *Communications of the ACM* (2018), 95–102.

[24] Gifford, D. K. Information Storage in a Decentralized Computer System. Tech. rep., Xerox Res. Cent., 1982.

[25] Green, M., and Miers, I. Bolt: Anonymous Payment Channels for Decentralized Currencies. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 473–489.

[26] Hong, C.-Y., Mandal, S., Al-Fares, M., Zhu, M., Alimi, R., Bhagat, C., Jain, S., Kaimal, J., Liang, S., Mendelev, K., et al. B4 and After: Managing Hierarchy, Partitioning, and Asymmetry for Availability and Scale in Google's Software-defined WAN. In *Conference of the ACM Special Interest Group on Data Communication* (2018), pp. 74–87.

[27] Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., and Vahdat, A. B4: Experience with a Globally-deployed Software Defined Wan. *ACM SIGCOMM* (2013).

[28] Kalodner, H., Goldfeder, S., Chen, X., Weinberg, S. M., and Felten, E. W. Arbitrum: Scalable, Private Smart Contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium* (2018), pp. 1353–1370.

[29] Khalil, R., and Gervais, A. Revive: Rebalancing Off-blockchain Payment Networks. In *ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 439–453.

[30] Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J. H., Lee, D., Wilkerson, C., Lai, K., and Mutlu, O. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *Annual International Symposium on Computer Architecture* (2014), pp. 361–372.

[31] Kocher, P., Horn, J., Fogh, A., , Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., Mangard, S., Prescher, T., Schwarz, M., and Yarom, Y. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy* (2019).

[32] Kogias, E. K., Jovanovic, P., Gailly, N., Khoffi, I., Gasser, L., and Ford, B. Enhancing Bitcoin Security and Performance with Strong Consistency via Collective Signing. In *USENIX Security Symposium* (2016).

[33] Kokoris-Kogias, E., Jovanovic, P., Gasser, L., Gailly, N., Syta, E., and Ford, B. OmniLedger: A Secure, Scale-out, Decentralized Ledger via Sharding. In *IEEE Symposium on Security and Privacy (SP)* (2018), pp. 583–598.

[34] Kosba, A., Miller, A., Shi, E., Wen, Z., and Papamanthou, C. Hawk: The Blockchain Model of Cryptography and Privacy-preserving Smart Contracts. In *IEEE symposium on security and privacy (SP)* (2016), pp. 839–858.

[35] Krupp, J., and Rossow, C. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium* (2018), pp. 1317–1333.

[36] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* (1978).

[37] Lamport, L., Shostak, R., and Pease, M. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1982).

[38] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., Mangard, S., Kocher, P., Genkin, D., Yarom, Y., and Hamburg, M. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium)* (2018).

[39] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. Making Smart Contracts Smarter. In *ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 254–269.

[40] Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., and Saxena, P. A Secure Sharding Protocol for Open Blockchains. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (2016), pp. 17–30.

[41] Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M., and Ravi, S. Concurrency and Privacy with Payment-channel Networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 455–471.

[42] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008.

[43] Peters, G. W., Panayi, E., and Chapelle, A. Trends in Crypto-currencies and Blockchain Technologies: A Monetary Theory and Regulation Perspective. *CoRR abs/1508.04364* (2015).

[44] Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., et al. The Design and Implementation of Open vSwitch. In *USENIX Symposium on Networked Systems Design and Implementation* (2015), pp. 117–130.

[45] Rekhter, Y., Li, T., and Hares, S. A Border Gateway Protocol 4 (BGP-4). Tech. rep., 2005.

[46] Schuster, F., Costa, M., Fournet, C., Gkantsidis, C., Peinado, M., Mainar-Ruiz, G., and Russinovich, M. VC3: Trustworthy data analytics in the cloud using SGX. In *IEEE Symposium on Security and Privacy (SP)* (2015), pp. 38–54.

[47] Sharples, M., and Domingue, J. The Blockchain and Kudos: A Distributed System for Educational Record, Reputation and Reward. In *Adaptive and Adaptable Learning* (2016), Springer International Publishing.

[48] Van Saberhagen, N. CryptoNote v 2.0. https://cryptonote.org/whitepaper.pdf, 2013.

[49] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* (2014).

[50] Zamani, M., Movahedi, M., and Raykova, M. RapidChain: Scaling Blockchain via Full Sharding. In *ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 931–948.

[51] Zhang, F., Cecchetti, E., Croman, K., Juels, A., and Shi, E. Town Crier: An Authenticated Data Feed for Smart Contracts. In *ACM SIGSAC Conference on Computer and Communications Security* (2016).

[52] Zheng, W., Dave, A., Beekman, J. G., Popa, R. A., Gonzalez, J. E., and Stoica, I. Opaque: An Oblivious and Encrypted Distributed Analytics Platform. In *USENIX Symposium on Networked Systems Design and Implementation* (2017), pp. 283–298.