HyperService: Interoperability and Programmability across Heterogeneous Blockchains

Anonymous Author(s)

ABSTRACT

Blockchain interoperability, which allows state transitions across different blockchain networks, is critical to facilitate major blockchain adoption. Existing interoperability protocols mostly focus on atomic token exchange between blockchains. However, as blockchains have been upgraded from just distributed ledgers into programmable state machines (thanks to smart contracts), the scope of blockchain interoperability goes beyond just token exchange. In this paper, we present HyperService, the first platform that delivers *interoperability* and *programmability* across *heterogeneous* blockchains. HyperService is powered by two innovative designs: (i) a developer-facing programming framework that allows developers to build cross-chain applications in a unified programming model; and (ii) a secure blockchain-facing cryptography protocol that provably realizes those applications on blockchains. We implement a prototype of HyperService in about 14,000 lines of code to demonstrate its practicality. Our experiment results show that HyperService imposes reasonable latency, in order of seconds, on the end-to-end execution of cross-chain applications.

1 INTRODUCTION

Over the last few years, we have witnessed rapid growth of several flagship blockchain applications, such as the payment system Bitcoin [52] and the smart contract platform Ethereum [24]. Since then, considerable amount of effort has been made to improve the performance of individual blockchains, including more efficient consensus algorithms [6, 18, 30, 41], improving transaction rate by sharding [19, 42, 50, 58] and payment channels [33, 36, 51], enhancing the privacy for smart contracts [26, 35, 43], and reducing their vulnerabilities via program analysis [22, 44, 49].

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 [2]. In a world deluged with isolated blockchains, interoperability is power. Blockchain interoperability is known to be invaluable to enable secure state transitions among different blockchains to make the decentralized world connected [23]. Existing interoperability proposals [20, 32, 34, 59] are mostly centering around atomic token exchange between two blockchains, aiming to eliminate the requirement of centralized exchanges. However, we argue that *token exchange is not the complete scope of blockchain interoperability*, because smart contracts executing on blockchains have essentially transformed blockchains from append-only distributed ledgers into programmable state machines. As a result, blockchain interoperability is complete only with programmability.

We recognize two categories of challenges for simultaneously delivering programmability and interoperability. First, the programming model of cross-chain decentralized applications (or dApps) is unclear. In general, from developers' perspective, it is desirable that cross-chain dApps could preserve the same state-machine-based programming abstraction as single-chain contracts [57]. This, however, raises a virtualization challenge to abstract away the heterogeneity of smart contracts on different blockchains so that the interactions and operations among those contracts can be *uniformly* specified when writing cross-chain dApps.

Further, the existing toke-exchange oriented interoperability protocol, such as atomic cross-chain swaps (ACCS) [1], is not generic enough to realize cross-chain dApps. This is because the "executables" of those dApps contain more complex operations than token transfers. For instance, our example dApp in § 2.3 requires to invoke a smart contract using parameters obtained from remote smart contracts deployed on different blockchains. The complexity of this programming operation is beyond mere token transfers.

To meet these challenges, we propose HyperService, the first platform for building and executing dApps across heterogeneous blockchains. At the very high level, HyperService is powered by two innovative designs: a developer-facing *programming framework* for writing cross-chain dApps, and the blockchain-facing cryptography protocol that securely realizes those dApps on underlying blockchains. Within the programming framework, we propose Unified State Model (USM), a blockchain-neutral and extensible model to describe cross-chain dApps, and the HSL, a high-level programming language to write cross-chain dApps under the USM programming model. dApps written in HSL are further compiled into HyperService executables which shall be executed by the underlying cryptography protocol.

UIP (short for universal inter-blockchain protocol) is the cryptography protocol that handles the complexity of actual cross-chain execution. UIP is (i) *generic*, operating on any blockchain with a public transaction ledger, (ii) *secure*, the executions of dApps either finish with verifiable correctness or abort due to security violations, where misbehaving parities are held accountable, and (iii) financially atomic, meaning all involved parities experience almost zero financial losses, regardless of the execution status of dApps. UIP is fully trustless, assuming no trusted entities.

Contributions. To the best of our knowledge, HyperService is the first platform that simultaneously offers *interoperability* and *programmability* across *heterogeneous* blockchains. Concretely, we make the following contributions.

(i) We propose the first programming framework for developing cross-chain dApps. The framework greatly facilitates dApp development by providing a virtualization layer on top of the underlying heterogeneous blockchains, yielding a unified model and a language to describe and program dApps. Using our framework, a developer without specialty in blockchain can easily write cross-chain dApps without implementing any cryptography.

(ii) We propose UIP, the first generic blockchain interoperability protocol whose design scope goes beyond realizing just cross-chain

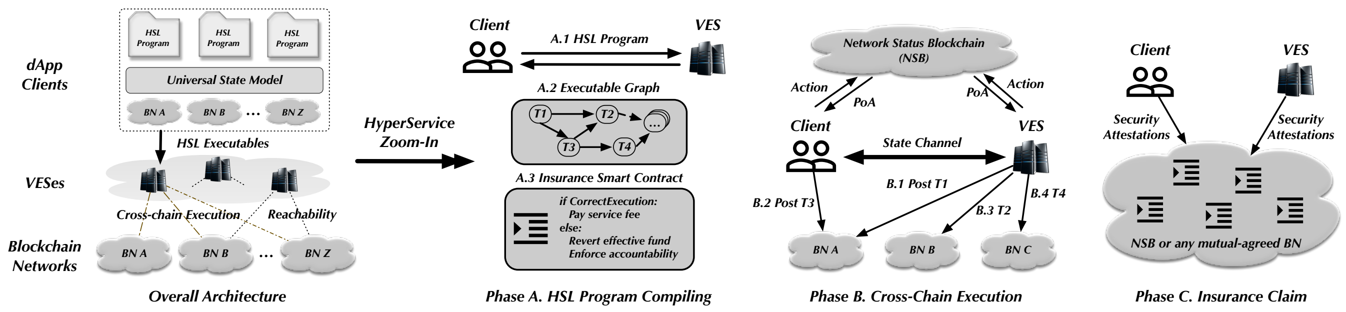

Figure 1: The architecture of HyperService.

token exchange. Rather, UIP is capable of securely realizing the complex cross-chain operations that involve smart contracts deployed on heterogeneous blockchains. We express the security properties of UIP via an ideal functionality $\mathcal{F}_{\text{UIP}}$ and rigorously prove that UIP realizes $\mathcal{F}_{\text{UIP}}$ in the Universal Composability (UC) framework [25].

(iii) We implement a prototype of HyperService in approximately 14,000 lines of code, and evaluate the prototype with three categories of cross-chain dApps. Our experiment results show that the end-to-end dApp execution latency on HyperService is in the order of seconds, and the aggregated throughout of HyperService surpasses the speed of Ethereum mainnet by three orders of magnitude, where further performance boost is possible via implementation optimizations.

2 HYPERSERVICE OVERVIEW

2.1 Architecture

Figure 1 depicts the architecture of HyperService. Architecturally, HyperService is designed around four components. (i) *dApp Clients* are the gateways for dApps to interact with the HyperService platform. When designing HyperService, we intentionally make clients to be lightweight, allowing both mobile and web applications to interact with HyperService. (ii) *Verifiable Execution Systems (VESes)* conceptually work as *blockchain drivers* that compile the high-level dApp programs given by the dApp clients into blockchain-executable transactions, which are the runtime executables on HyperService. VESes and dApps employ the underlying UIP cryptography protocol to securely execute those runtimes across different blockchains. UIP itself has two building blocks: (iii) the Network Status Blockchain (NSB) and (iv) the Insurance Smart Contracts (ISCs). The NSB, conceptually, is a *blockchain of blockchains* built by HyperService to provide an objective and unified view of the execution status dApps, based on which the ISCs arbitrate the correctness or violation of dApp executions. In case of exceptions, the ISCs financially revert all executed transactions to guarantee financial atomicity and hold misbehaved entities accountable.

2.2 Universal State Model

A blockchain, as well as smart contracts (or dApps) executed on the blockchain, is often perceived as a state machine [57]. We desire to preserve the similar abstraction for developers when writing cross-chain dApps. Towards this end, HyperService proposes USM, a blockchain-neutral and extensible model for describing state transitions across different blockchains, which in essential defines cross-chain dApps. USM realizes a virtualization layer to unify the underlying heterogeneous blockchains. Such virtualization includes: (i) blockchains, regardless of their implementations (*e.g.,* consensus mechanisms, smart contract execution environment, programming languages and so on), are abstracted as *objects* with public state variables and functions; (ii) developers program dApps by specifying desired operations over those objects, along with the relative ordering among those operations, as if all the objects were local to a single machine.

Formally, USM is defined as $\mathcal{M} = \{\mathcal{E}, \mathcal{P}, \mathcal{C}\}$ where $\mathcal{E}$ is a set of *entities*, $\mathcal{P}$ is a set of *operations* performed over those entities, and $\mathcal{C}$ is a set of constraints defining the *dependency* of those operations. Entities are to describe the objects abstracted from blockchains. All entities are conceptually local to $\mathcal{M}$, regardless of which blockchains they are obtained from. Entities come with *kinds*, and each entity kind has different attributes. The current version of USM defines two concrete kinds of entities, *accounts* and *contracts*, as tabulated in Table 1. Specifically, an account entity is associated with a uniquely identifiable address, as well as its balance in certain units. A contract entity, besides its address, is further associated with a list of state variables, callable interfaces and source code. Entity attributes are crucial to enforce the security and correctness of dApps during compilation, as discussed in § 2.3.

An operation in USM, conceptually, defines a step of computation performed over several entities when writing dApps. Table 1 lists two concrete kinds of operations in USM: a *payment* operation that describes the balance updates between two account entities at a certain exchange rate; an *invocation* operation that describes the execution of an interface of a contract entity using compatible parameters, whose values could be obtained from other contract entities' state variables.

Although operations are conceptually local, each operation is eventually compiled into one or more transactions on different blockchains, whose executions are not synchronized. To honor the possible dependency among events in distributed computing [45], USM, therefore, defines constraints to specify dependencies among operations. Currently, USM supports two kinds of dependencies, precondition and deadline, where an operation can proceed only if all its preconditioning operations are finished, and an operation must proceed within a bounded period of time after its dependencies are satisfied. Preconditions and deadlines offer desirable programming abstraction for dApps: (i) preconditions enable developers to organize their operations into a directed acyclic graph,

Table 1: Example of Entities, Operations and Dependency in USM

Entity Kind	Attributes	*Operation Kind*	Attributes	*Dependency Kind*
account	address, balance, unit	*payment*	from, to, value, exchange rate	*precondition*
contract	address, StateVariable[], Interfaces[], source	*invocation*	interface, parameters[const, Contract.SV, ...], invoker	*deadline*

where the state of upstream nodes is persistent and can be used by downstream nodes; (ii) deadlines are crucial to ensure forward progress of executions.

2.3 HyperService Programming Language

To demonstrate the usage of USM, we develop HSL, a programming language to write cross-chain dApps under the modeling of USM.

2.3.1 An Introductory Example for HSL Program

Financial derivatives are among the most commonly cited smart contract applications. Many financial derivatives often rely on authentic data feed, typically known as an *oracle*, as inputs. For instance, a standard call Option contract needs to have a genuine strike price. Existing oracles [3, 60] require a smart contract on the blockchain to serve as the front-end to interact with other client smart contracts. As a result, it is difficult to build a dependable and unbiased oracle that is simultaneously accessible to multiple blockchains, because we cannot simply deploy an oracle front-end smart contract on each individual blockchain since synchronizing the execution of those front-end contracts requires blockchain interoperability, *i.e.,* we see a chicken-and-egg problem. This limitation, in turn, prevents dApps from spreading their business across multiple blockchains. For instance, a call option contract deployed on Ethereum forces investors to exercise the option using Ether, but not in other cryptocurrency.

As an introductory example, we shall see how conceptually simple, yet elegant, it is, from developers' perspective, to build a universal call-option dApp that allows investors to natively exercise options with the cryptocurrency they prefer. The code snippet shown Figure 2 is the HSL program for implementing such a dApp. In this dApp, both option contracts deployed on blockchains *ChainY* and *ChainZ* rely on the same Broker contract on *ChainX* to provide the strike price, as specified by the *invocation* operations in lines 14 and 15 of Figure 2. Detailed HSL grammar is given in Grammar 1.

2.3.2 HSL Program Compilation

The core of HyperService programming framework is the HSL compiler. The compiler performs two major tasks: (i) enforcing security and correctness check on HSL programs and (ii) compiling HSL programs into blockchain-executable transactions.

One of the key differentiations of HyperService is that it allows dApps to natively define interactions and operations among smart contracts deployed on heterogeneous blockchains. Since these smart contracts could be written in different languages, HSL provides a multi-language front end to analyze the source code of those contract contracts. It extracts the type information of their public state variables and functions, and then converts them into the unified types defined by HSL (§ 3.1). This enables effective correctness check on the HSL programs (§ 3.3). For instance, it ensures that all the parameters used in a contract *invocation* operation are compatible and verifiable, even if these arguments are extracted from remote contracts written in different languages compared to that of the invoking contract.

```
1  # Import the source code of contracts written in different languages.
2  import ("broker.sol", "option.vy", "option.go")
3  # Entity definition.
4  # Attributes of a contract entity are implicit from its source code.
5  account a1 = ChainX::Account(0x7019..., 100, xcoin)
6  account a2 = ChainY::Account(0x47a1..., 0, ycoin)
7  account a3 = ChainZ::Account(0x61a2..., 50, zcoin)
8  contract c1 = ChainX::Broker(0xbba7...)
9  contract c2 = ChainY::Option(0x917f...)
10 contract c3 = ChainZ::Option(0xefed...)
11 # Operation definition.
12 op op1 invocation c1.GetStrikePrice() using a1
13 op op2 payment 50 xcoin from a1 to a2 with 1 xcoin as 0.5 ycoin
14 op op3 invocation c2.CashSettle(10, c1.StrikePrice) using a2
15 op op4 invocation c3.CashSettle(5, c1.StrikePrice) using a3
16 # Dependency definition.
17 op1 before op2, op4; op3 after op2
18 op1 deadline 10 blocks; op2, op3 deadline default; op4 deadline 20 mins
```

Figure 2: A cross-chain Option dApp written in HSL.

Once a HSL program passes the syntax and correctness check, the compiler will generate an *executable* for the program. The executable is structured in form of a Transaction Dependency Graph, which contains (i) the complete information for computing a set of blockchain-executable transactions, (ii) useful metadata for each transaction that is required for correct execution, and (iii) the preconditions and deadlines of those transactions that honor the dependency constraints specified in the HSL program (§ 3.4).

In HyperService, the Verifiable Execution Systems (VESes) are the actual entities that own the HSL compiler and therefore resume the aforementioned compiler responsibilities. Because of this, VESes work as *blockchain drivers* that bridge our high-level programming framework with the underlying blockchains. Each VES is a distributed system providing trust-free service to compile and execute HSL programs given by dApp clients. VESes are trust-free because their actions taken during dApp executions are verifiable. Each VES defines its own service model, including reachability (*i.e.,* the set of blockchains that the VES supports), service fee charged for correct executions, and insurance plan (*i.e.,* the expected compensation to dApps if the VES's execution is proven to be incorrect). dApps have full autonomy to select VESes that satisfy their requirements. In Appendix A.3, we discuss the anatomy of VESes.

Besides owning the HSL compiler, VESes also participate in the actual execution of HSL executables, as discussed below.

2.4 Universal Inter-Blockchain Protocol

To correctly execute a dApp, all the transactions in its executable must be posted on blockchains for execution, and meanwhile their preconditions and deadlines are honored. Although this executing procedure is conceptually simple (thanks to the HSL abstraction), it is very challenging to enforce correct executions in a fully trust-free manner where (i) no trusted authority is allowed to coordinate the execution on underlying blockchains and (ii) no mutual trust between VESes and dApp clients are established.

To address this challenge, HyperService designs UIP, a cryptography protocol between VESes and dApp clients to securely execute HSL executables on blockchains. UIP can work on any blockchain with public ledgers, imposing no additional requirements such as their consensus protocols and code execution environment. UIP provides strong security guarantee for executing dApps such that dApps are correctly executed only if the correctness is publicly verifiable by all stakeholders; otherwise, UIP holds the misbehaving parties accountable, and financially reverts all committed transactions to achieve financial atomicity.

UIP is powered by two innovative designs: the Network Status Blockchain (NSB) and the Insurance Smart Contract (ISC). The NSB is a blockchain designed by HyperService to provide objective and unified views on the status of dApp executions. On the one hand, the NSB consolidates the committed transactions of all underlying blockchains into Merkle trees, providing unified representations for transaction status in form of verifiable Merkle proofs. On the other hand, the NSB supports Proofs of Actions (PoAs), allowing both dApp clients and VESes to construct proofs to certify their actions taken during cross-chain executions. The ISC is a code-arbitrator. It takes input as transaction status Merkle proofs constructed from the NSB to determine the correctness or violation of dApp executions, and meanwhile uses action Merkle proofs to determine the accountable parities in case of violations.

In § 4.6, we define the security properties of UIP via an ideal functionality and then rigorously prove that UIP realizes the ideal functionality in UC-framework [25].

2.5 Assumption and Threat Model

We assume that the cryptographic primitives and the consensus protocol of all underlying blockchains are secure, although we impose no constraints on their consensus efficiency. For each blockchain, we assume that transaction finality on the blockchain can be securely guaranteed. On Nakamoto consensus based blockchains, this is achieved by assuming that the probability of blockchain reorganizations drops exponentially as new blocks are appended (*common-prefix property*) [31]. On Byzantine tolerance based blockchains [14], finality is guaranteed by signatures from a quorum of permissioned consensus nodes. We also assume that each underlying blockchain has a public ledger that allows external parties to exam and prove transaction finality, as well as the *public* state of smart contracts.

The correctness of UIP relies on the correctness of the NSB. The NSB is best implemented as a permissioned blockchain where any information on NSB becomes legitimate only if a quorum of consensus nodes have approved the information. We thus assume that at least $\mathcal{K}$ consensus nodes that maintain the NSB are honest, where $\mathcal{K}$ is the quorum threshold (*e.g.,* the majority).

We consider a Byzantine adversary that interferes with our protocol arbitrarily, including delaying and reordering network messages indefinitely, and compromising protocol participants. Should at least one participant not be compromised by the adversary, the security properties of UIP are guaranteed.

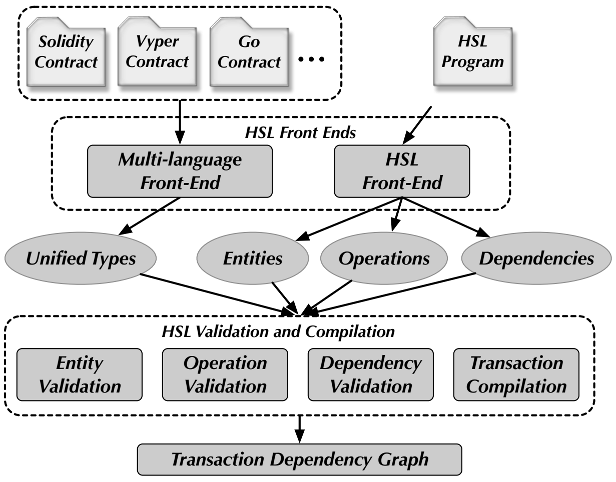

Figure 3: Workflow of HSL Compilation.

3 PROGRAMMING FRAMEWORK

The design of HyperService programming framework centers around the HSL compiler. Figure 3 depicts the compilation workflow. The HSL compiler has two frond-ends: one for extracting entities, operations, and dependencies from a HSL program and one for extracting public state variables and methods from smart contracts deployed on blockchains. A unified type system is designed to ensure that smart contracts written in different languages can be abstracted as the interoperable entities defined in the HSL program. Afterwards, the compiler performs semantic validations on all entities, operations and dependencies to ensure the security and correctness of the HSL program. Finally, the compiler produces an executable for the HSL program, which is structured in form of a transaction dependency graph. We next describe the details of each component.

3.1 Unified Type System

The primary goal of proposing the USM is to provide a unified virtualization layer for developers to define *invocation* operations in their HSL programs, without handling the heterogeneity of contract entities. Towards this end, the programming framework internally defines a Unified Type System so that the state variables and methods of all contract entities can be abstracted using the unified types when writing HSL programs. This enables the HSL compiler to ensure that all arguments specified in an *invocation* operation are *compatible* (§ 3.3).

Specifically, the unified type system defines nine elementary types, including *Boolean*, *Numeric*, *Address*, *String*, *Array*, *Map*, *Struct*, *Function*, *Contract*. The data types that are commonly used in smart contract programming languages will be *mapped* into these unified types during compilation. For example, Solidity does not fully support fixed-point number, but Vyper (*decimal*) and Go (*float*) do. Also, Vyper's string is fixed-sized (declared as string[*Integer*]), but Solidity's string is dynamically-sized (declared as string). Our multi-lang front-end recognizes these differences and performs type conversion to map all the numeric literals including integers

Table 2: Unified Type Mapping for Solidity, Vyper, and Go

Type	*Solidity*	*Vyper*	*Go*
Boolean	bool	bool	bool
Numeric	int, uint	int128, uint256, decimal, unit type	int, uint, uintptr, float
Address	address	address	string
String	string	string	string
Array	array, bytes	array, bytes	array, slice
Map	mapping	map	map
Struct	struct	struct	struct
Function	function, enum	def	func
Contract	Contract	file	type

and decimals to the *Numeric* type, and the strings to the *String* type. For types that are similar in Solidity, Vyper, and Go, such as *Boolean*, *Map*, and *Struct*, we simply map them to the corresponding type in our unified type. Finally, Solidity and Vyper provide special types for representing contract addresses, which are mapped to the *Address* type. But Go does not provide a type for contract addresses, and thus Go's *String* type is mapped to the *Address* type. The mapping of language-specific types to the unified type system is tabulated in Table 2. The unified type system is horizontally scalable to support additional strong-typed programming languages. Note that using complex data types as contract function parameters has not been fully supported yet on many blockchains. We thus leave the support for complex types in HSL to future work.

3.2 HSL Language Design

The language constructs provided by HSL are coherent with the USM, allowing developers to straightforwardly specify entities, operations, and dependencies in HSL programs. One additional construct, *import*, is added to import the source code of contract entities, as discussed below. Grammar 1 shows the representative rules of HSL. We omit the terminal symbols such as ⟨*id*⟩ and ⟨*address*⟩.

Contract Importing. Developers use the ⟨*import*⟩ rule to include the source code of contract entities. Depending on the programming language of an imported contract, HSL's multi-lang front end uses the corresponding parser to parse the source code, based on which it performs semantic validation (§ 3.3). For security purpose, the compiler should verify that the imported source code is consistent with the actual deployed code on blockchain, for instance, by comparing their compiled byte code.

Entity Definition. The ⟨*entity_def*⟩ rule specifies the definition of an *account* or a *contract* entity. An entity is defined via constructor, where the on-chain (⟨*address*⟩) of the entity is a required parameter. An *account* entity can be initialized with an optional unit (⟨*unit*⟩) to specify the cryptocurrency held by the account. All *contract* entities must have the corresponding contract objects/classes in one of the imported source code files. Each entity is assigned with a name (⟨*entity_name*⟩) that can be used for defining operations.

Operation Definition. The ⟨*op_def*⟩ rule specifies the definition of a *payment* or an *invocation* operation. A *payment* operation (⟨*op_payment*⟩) specifies the transfer of a certain amount of coins (⟨*coin*⟩) between two accounts that may live on different blockchains (⟨*accts*⟩). The ⟨*exchange*⟩ rule is used to specify the exchange rate between the coins held by the two accounts. An *invocation* operation (⟨*op_invocation*⟩) specifies calling one contract entity's public method with certain arguments (⟨*call*⟩). The arguments passed to a method invocation can be literals (⟨*int*⟩, ⟨*float*⟩, ⟨*string*⟩), and state variables (⟨*state_var*⟩) of other contract entities. When using state variables, semantic validation is required (§ 3.3).

```
⟨hsl⟩           ::= (⟨import⟩)+ (⟨entity_def⟩)+ (⟨op_def⟩)+ (⟨dep_def⟩)*
Contract Imports:
⟨import⟩        ::= 'import' '(' ⟨file⟩ (',' ⟨file⟩)* ')'
⟨file⟩          ::= ⟨string⟩

Entity Definition:
⟨entity_def⟩    ::= ⟨entity_type⟩ ⟨entity_name⟩ '=' ⟨chain_name⟩ '::'
                    ⟨constructor⟩
⟨entity_name⟩   ::= ⟨id⟩
⟨chain_name⟩    ::= 'Chain' ⟨id⟩
⟨constructor⟩   ::= ⟨contract_type⟩ '(' ⟨address⟩, (⟨unit⟩)? ')'
⟨contract_type⟩ ::= 'Account' | ⟨id⟩
⟨entity_type⟩   ::= 'account' | 'contract'

Operation Definition:
⟨op_def⟩        ::= ⟨op_payment⟩ | ⟨op_invocation⟩
⟨op_payment⟩    ::= 'op' ⟨op_name⟩ 'payment' ⟨coin⟩ ⟨accts⟩ ⟨exchange⟩
⟨op_name⟩       ::= ⟨id⟩
⟨coin⟩          ::= ⟨num⟩ ⟨unit⟩
⟨accts⟩         ::= 'from' ⟨acct⟩ 'to' ⟨acct⟩
⟨acct⟩          ::= ⟨id⟩
⟨exchange⟩      ::= 'with' ⟨coin⟩ 'as' ⟨coin⟩
⟨op_invocation⟩ ::= 'op' ⟨op_name⟩ 'invocation' ⟨call⟩ 'using' ⟨acct⟩
⟨call⟩          ::= ⟨recv⟩ '.' ⟨method⟩ '(' (arg)* ')'
⟨arg⟩           ::= ⟨int⟩ | ⟨float⟩ | ⟨string⟩ | ⟨state_var⟩
⟨state_var⟩     ::= ⟨varname⟩ '.' ⟨prop⟩

Dependency Definition:
⟨dep_def⟩       ::= ⟨temp_deps⟩ | ⟨del_deps⟩
⟨temp_deps⟩     ::= ⟨temp_dep⟩ (';' ⟨temp_dep⟩)*
⟨temp_dep⟩      ::= ⟨op_name⟩ ('before' | 'after') ⟨op_name⟩ (','
                    ⟨op_name⟩)*
⟨del_deps⟩      ::= ⟨del_dep⟩ (';' ⟨del_dep⟩)*
⟨del_dep⟩       ::= ⟨op_name⟩ (',' ⟨op_name⟩)* 'deadline' ⟨del_spec⟩
⟨del_spec⟩      ::= ⟨int⟩ 'blocks' | 'default' | ⟨int⟩ ⟨time_unit⟩
```

Grammar 1: Representative BNF grammar of HSL

Dependency Definition. The ⟨*dep_def*⟩ specifies the rule of defining preconditions and deadlines for operations. A *precondition* (⟨*temp_deps*⟩) specifies the temporal constraints for the execution order of operations. A *deadline* (⟨*del_deps*⟩) specifies the deadline constraint of each operation. The deadline dependency may be given either in absolute time (⟨*int*⟩ ⟨*time_unit*⟩) or via the number of blocks on NSB (⟨*int*⟩ *blocks*), as explained in § 3.4.

3.3 Semantic Validation

The compiler performs two types of semantic validation to ensure the security and correctness of HSL programs. First, the compiler guarantees the *compatibility* and *verifiability* of the arguments used in *invocation* operations, especially when those arguments are obtained from other contract entities. For compatibility check, the compiler performs type checking to ensure the types of arguments and the types of method parameters are mapped to the same unified type. For verifiability check, the compiler ensures that only literals and state variables that are publicly stored on blockchains are eligible to be used as arguments in *invocation* operations. For example, the return values of method calls to a contract entity are not eligible if these results are not persistent on blockchains. This requirement is necessary for the UIP protocol to construct publicly verifiable attestations to prove that correct values are used to invoking contracts during actual on-chain execution. Second, the compiler performs dependency validation to make sure that the dependency constraints defined in a HSL program uniquely specify a directed cycle graph connecting all operations. This ensures that no conflicting temporal constraints are specified.

3.4 HSL Program Executables

Once a HSL program passes all validations, the HSL compiler generates executables for the program in form of a transaction dependency graph $\mathcal{G}_T$. The vertices of $\mathcal{G}_T$ are referred to as *transaction wrappers* , each of which contains the complete information to compute an on-chain transaction executable on a specific blockchain, as well as additional metadata associated with the transaction. The edges in $\mathcal{G}_T$ define the preconditioning requirements among transactions, which are consistent with the dependency constraints specified by the HSL program. Figure 4 depicts the $\mathcal{G}_T$ generated for the HSL program in Figure 2.

A transaction wrapper is in form of $\mathcal{T} := [\text{from}, \text{to}, \text{seq}, \text{meta}]$, where the pair <from, to> decides the sending and receiving addresses of the on-chain transaction, seq (omitted in Figure 4) represents the sequence number of $\mathcal{T}$ in $\mathcal{G}_T$, and meta stores the structured and customizable metadata for $\mathcal{T}$. Below we explain the fields of meta. First, to achieve financial atomicity, meta must populate a tuple $\langle \text{amt}, \text{dst} \rangle$ for fund reversion. In particular, amt specifies the total value that the *from* address has to spend when $\mathcal{T}$ is committed on its destination blockchain, which includes both the explicitly paid value in $\mathcal{T}$, as well as any gas fee. If the entire execution fails with exceptions whereas $\mathcal{T}$ is committed, the dst account is guaranteed to receive the amount of fund specified in amt. As we shall see in § 4.4, the fund reversion is handled by the Insurance Smart Contract (ISC). Therefore, the unit of amt (represented as *ncoin* in Figure 4) is given based on the cryptocurrency used by the blockchain where the ISC is deployed, and the dst should live on the hosting blockchain as well.

Second, for a transaction (such as T1) whose resulting state is subsequently used by other downstream transactions (such as T4), its meta needs to be populated with a corresponding state proof. This proof should be collected from the transaction's destination blockchain after the transaction is finalized (*c.f.,* § 4.2.3). Third, a cross-chain payment operation in the HSL program results in multiple transactions in $\mathcal{G}_T$. For instance, to realize the op1 in Figure 2, two individual transactions, involving the *relay accounts* owned by the VES, are generated. As blockchain drivers, each VES is supposed to own some accounts on all blockchains that it has visibility so that the VES is able to send and receive transactions on those blockchains. For instance, in Figure 4, the *relayX* and *relayY* are two accounts used by the VES to bridge the balance updates between *ChainX::a1* and *ChainY::a2*. Because of those VES-owned accounts, $\mathcal{G}_T$ is typically VES specific.

Finally, besides using absolute timestamps, the deadlines of transactions could be specified based on the number of blocks on the NSB. This is because the NSB constructed a unified view of the status of all underlying blockchains and therefore can measure the execution time of each transaction (*c.f.,* § 4.2.2).

In summary, the executable produced by the HSL complier defines the blueprint of cross-blockchain execution to realize the HSL program. It is the input instructions that direct the underlying cryptography protocol UIP, as detailed below.

4 UIP DESIGN DETAIL

UIP is the cryptography protocol that executes HSL program executables. The main protocol Prot_{UIP} is divided into *five* preliminary protocols. In particular, Prot_{VES} and Prot_{CLI} define the execution protocols implemented by VESes and dApp clients, respectively. Prot_{NSB} and Prot_{ISC} are the protocol realization of the NSB and ISC, respectively. Lastly, Prot_{UIP} includes Prot_{BC}, the protocol realization of a general-purposed blockchain. Overall, Prot_{UIP} has two phases: the execution phase where the transactions specified in the

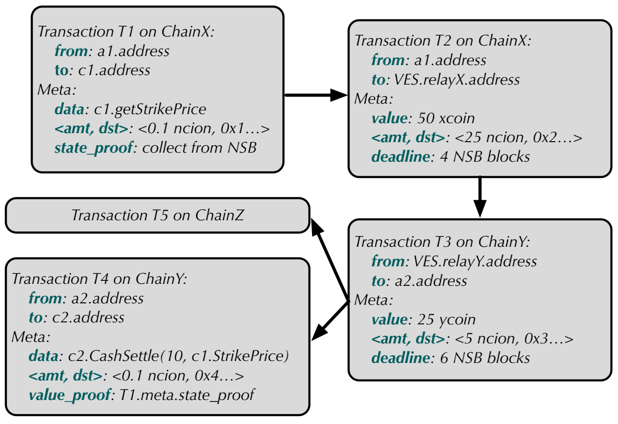
Transaction T1 on ChainX:
from: a1.address
to: c1.address
Meta:
data: c1.getStrikePrice
<amt, dst>: <0.1 ncion, 0x1...>
state_proof: collect from NSB
Transaction T2 on ChainX:
from: a1.address
to: VES.relayX.address
Meta:
value: 50 xcoin
<amt, dst>: <25 ncion, 0x2...>
deadline: 4 NSB blocks
Transaction T5 on ChainZ
Transaction T3 on ChainY:
from: VES.relayY.address
to: a2.address
Meta:
value: 25 ycoin
<amt, dst>: <5 ncion, 0x3...>
deadline: 6 NSB blocks
Transaction T4 on ChainY:
from: a2.address
to: c2.address
Meta:
data: c2.CashSettle(10, c1.StrikePrice)
<amt, dst>: <0.1 ncion, 0x4...>
value_proof: T1.meta.state_proof

Figure 4: $\mathcal{G}_T$ generated for the example HSL program.

HSL executables are posted on blockchains and the insurance claim phase where the execution correctness or violation is arbitrated.

4.1 Protocol Preliminaries

4.1.1 *Runtime Transaction State*

During the execution phase, a transaction may be in any of the following state {unknown, init, inited, open, opened, closed}, where a latter state is considered more *advanced* than a former one. The state of each transaction must be gradually promoted following the above sequence. For each state (expect the unknown), Prot_{UIP} produces a corresponding attestation to prove the state. When the execution phase terminates, the final execution status of the HSL program is collectively decided by the state of all transactions, based on which Prot_{ISC} arbitrates its correctness or violation.

4.1.2 *Off-chain State Channels*

The protocol exchange between Prot_{VES} and Prot_{CLI} can be conducted via off-chain state channels for low latency. One challenge, however, is that it is difficult to enforce accountability for non-closed transactions without preserving the execution steps by both parties. To address this issue, Prot_{UIP} proposes Proof of Actions (PoAs), allowing Prot_{VES} and Prot_{CLI} to stake their execution steps on NSB. As a result, the NSB is treated as a publicly-observable *fall-back* communication medium for the off-chain channel. The benefit of this dual-medium design is that the protocol exchange between Prot_{VES} and Prot_{CLI} can still proceed agilely via off-chain channels in typical scenarios, whereas the full granularity of their protocol exchange is preserved on the NSB, eliminating the ambiguity for Prot_{ISC} to decide the accountable party in case of exceptions.

As mentioned in § 4.1.1, Prot_{UIP} produces security attestations to prove the runtime state of transactions. As we shall see below, an attestation may come in two forms: a certificate, denoted by Cert, signed by Prot_{VES} or/and dApp during their off-chain exchange, or an *on-chain* Merkle proof, denoted by Merk, constructed using the NSB and underlying blockchains. An Cert and its corresponding Merk are treated equivalently by the Prot_{ISC} in code arbitration.

4.1.3 *Architecture of the NSB*

The NSB is a blockchain designed to provide an objective view on the execution status of dApps. Figure 5 depicts the architecture of NSB blocks. The NSB is a fully functional blockchain. Thus,

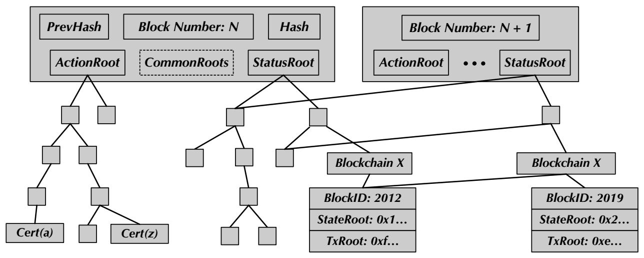

Figure 5: The architecture of NSB blocks.

its block, similar to that of other blockchains, contains several typical fields, such as the hash fields to link blocks together and the Merkle trees to store transactions and state. To support the extra functionality of the NSB, an NSB block contains two additional Merkle Tree roots: StatusRoot and ActionRoot.

StatusRoot is the root of a Merkle tree (referred as StatusMT) that stores *transaction status* of underlying blockchains. The NSB represents the transaction status of a blockchain using the TxRoots and StateRoots retrieved directly from the blockchain's public ledger. Although the exact namings may vary on different blockchains, in general, a TxRoot and StateRoot of a blockchain block represent the root of a Merkle tree storing transactions and storage state (*e.g.,* account balance, contract state). Note that the NSB only stores *relevant* blockchain state, where a blockchain block is considered to be relevant if it packages at least one transaction that is part of any dApp executables.

ActionRoot is the root of a Merkle tree (referred to as ActionMT) whose leaf nodes store certificates computed by VESes and dApp clients. Each certificate represents a certain step taken by either the VES or the dApp client during the execution phase. To prove such an action, a party simply needs to construct a Merkle proof to demonstrate that the certificate mapped to the action can be linked to a committed block on the NSB. These PoAs are crucial for the ISC to enforce accountability if the execution fails. Since the information under each ActionMT is static, we can lexicographically sort the ActionMT to achieve fast search, as well as convenient proof of non-membership.

Note that the construction of StatusMT ensures that each blockchain can have a dedicated subtree for storing its transaction status. This makes the NSB *trivially shardable on the granularity of each blockchain*, ensuing that the NSB won't become the choke point as HyperService continuously incorporates more blockchains. As we shall see in § 4.5, $\mathsf{Prot}_{\mathsf{NSB}}$ is the protocol that specifies the detailed construction of both roots and guarantees their correctness.

4.2 Execution Protocol by VESes

The full protocol of $\mathsf{Prot}_{\mathsf{VES}}$ is detailed in Figure 6. Below we clarify some technical subtleties.

4.2.1 Post Compilation and Session Setup

After $\mathcal{G}_T$ is generated, $\mathsf{Prot}_{\mathsf{VES}}$ initiates an execution session for $\mathcal{G}_T$ in the PostCompiliation daemon. The primary goal of the initialization is to create and deploy an insurance contract to protect the execution of $\mathcal{G}_T$. Towards this end, $\mathsf{Prot}_{\mathsf{VES}}$ interacts with the protocol $\mathsf{Prot}_{\mathsf{ISC}}$ to create the insurance *contract* for $\mathcal{G}_T$, and further deploys the *contract* on NSB after the dApp client $\mathcal{D}$ agrees on the *contract*. Throughout the paper, $\mathsf{Cert}([*];\mathsf{Sig})$ represents a signed certificate proving that the signing party agrees on the value enclosed in the certificate. We use $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$ and $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$ to represent the signature by $\mathsf{Prot}_{\mathsf{VES}}$ and $\mathsf{Prot}_{\mathsf{CLI}}$, respectively.

Additionally, both $\mathsf{Prot}_{\mathsf{VES}}$ and $\mathsf{Prot}_{\mathsf{CLI}}$ are required to deposit sufficient funds to $\mathsf{Prot}_{\mathsf{ISC}}$ to ensure that $\mathsf{Prot}_{\mathsf{ISC}}$ holds sufficient funds to financially revert all committed transactions regardless of the step at which the execution aborts prematurely. Intuitively, each party would need to stake at least the total amount of incoming funds to the party *without* deducting the outgoing funds. This strawman design, however, require high stakes. More desirably, considering the dependency requirements in $\mathcal{G}_T$, an party $\mathcal{X}$ ($\mathsf{Prot}_{\mathsf{VES}}$ or $\mathsf{Prot}_{\mathsf{CLI}}$) only needs to stake

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

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 $\mathcal{T}.\mathsf{from} = \mathsf{Prot}_{\mathsf{VES}}$ or $\mathcal{T}$ is originated from $\mathsf{Prot}_{\mathsf{VES}}$, we mean that $\mathcal{T}$ is sent and signed by an account owned by $\mathsf{Prot}_{\mathsf{VES}}$. Likewise, $\mathcal{T}.\mathsf{from} = \mathsf{Prot}_{\mathsf{CLI}}$ indicates that $\mathcal{T}$ is sent from an account entity defined in the HSL program. $\mathsf{Prot}_{\mathsf{ISC}}$ refunds any remaining funds after the contract is terminated.

After the *contract* is instantiated and sufficiently staked, $\mathsf{Prot}_{\mathsf{VES}}$ initializes its internal bookkeeping for the session. The two notations S_{Cert} and S_{Merk} represent two sets that store the signed certificates received via off-chain channels and on-chain Merkle proofs constructed using $\mathsf{Prot}_{\mathsf{NSB}}$ and $\mathsf{Prot}_{\mathsf{BC}}$.

4.2.2 Protocol Exchange for Transaction Handling

In $\mathsf{Prot}_{\mathsf{VES}}$, SInitedTrans and OpenTrans are two handlers processing *northbound* transactions which originates from $\mathsf{Prot}_{\mathsf{VES}}$. The SInitedTrans handling for $\mathcal{T}$ is invoked when all its preconditions are finalized, which is detected by the watching service of $\mathsf{Prot}_{\mathsf{VES}}$ (*c.f.,* § 4.2.3). The SInitedTrans computes $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ to prove $\mathcal{T}$ is in the inited state , and then passes it to the corresponding handler of $\mathsf{Prot}_{\mathsf{CLI}}$ for subsequent processing. Meanwhile, SInitedTrans stakes $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ on $\mathsf{Prot}_{\mathsf{NSB}}$, and later it retrieves a Merkle proof $\mathsf{Merk}^{i}_{\mathcal{T}}$ from the NSB to prove that $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ has been sent. $\mathsf{Merk}^{\mathsf{id}}_{\mathcal{T}}$ essentially is a hash chain linking $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ back to an ActionRoot on a committed block of the NSB. The proof retrieval is a non-blocking operation triggered by the consensus update on the NSB.

The OpenTrans handler pairs with SInitedTrans. It listens for a timestamped $\mathsf{Cert}^{o}_{\mathcal{T}}$, which is supposed to be generated by $\mathsf{Prot}_{\mathsf{CLI}}$ after it processes $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ from $\mathsf{Prot}_{\mathsf{VES}}$. OpenTrans performs special correctness check on the $\mathsf{ts}_{\mathsf{open}}$ enclosed in $\mathsf{Cert}^{o}_{\mathcal{T}}$. In particular, $\mathsf{Prot}_{\mathsf{VES}}$ and $\mathsf{Prot}_{\mathsf{CLI}}$ use the block height of the NSB as a calibrated clock. By checking that $\mathsf{ts}_{\mathsf{open}}$ is within a bounded range of the NSB height, $\mathsf{Prot}_{\mathsf{VES}}$ ensures that the $\mathsf{ts}_{\mathsf{open}}$ add by $\mathsf{Prot}_{\mathsf{CLI}}$ is fresh. After all correctness checks on $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ are passed, the state of $\mathcal{T}$ is promoted from open to opened. OpenTrans then computes certificate to prove the updated state and posts $\widetilde{T}$ on its destination blockchain for on-chain execution. Throughout the paper, $\widetilde{T}$ denotes the on-chain executable transaction computed and signed using the information contained in $\mathcal{T}$. Note that the difference between

```
Init: Data := $\emptyset$
Daemon PostCompilation():
  generate the session ID sid $\leftarrow \{0,1\}^{\lambda}$
  call [cid, *contract*] := Prot$_{\mathsf{ISC}}$.CreateContract($\mathcal{G}_{\mathcal{T}}$)
  send Cert([sid, $\mathcal{G}_{\mathcal{T}}$, *contract*]; $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$) to Prot$_{\mathsf{CLI}}$ for approval
  halt until Cert([sid, $\mathcal{G}_{\mathcal{T}}$, *contract*]; $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$) is received
  package *contract* as a valid transaction $\widetilde{contract}$
  call Prot$_{\mathsf{NSB}}$.Exec($\widetilde{contract}$) to deploy the $\widetilde{contract}$
  halt until $\widetilde{contract}$ is initialized on Prot$_{\mathsf{NSB}}$
  call Prot$_{\mathsf{ISC}}$.StakeFund to stake the required funds in Prot$_{\mathsf{ISC}}$
  halt until $\mathcal{D}$ has staked its required funds in Prot$_{\mathsf{ISC}}$
  initialize Data[sid] := {$\mathcal{G}_{\mathcal{T}}$, cid, $\mathcal{S}_{\mathsf{Cert}}$=$\emptyset$, $\mathcal{S}_{\mathsf{Merk}}$=$\emptyset$}
Daemon Watching(sid, {Prot$_{\mathsf{BC}}$, ...}) private:
  ($\mathcal{G}_{\mathcal{T}}$, _, $\mathcal{S}_{\mathsf{Cert}}$, $\mathcal{S}_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  for each $\mathcal{T} \in \mathcal{G}_{\mathcal{T}}$ :
    continue if $\mathcal{T}$.state is not opened
    identify $\mathcal{T}$'s on-chain counterpart $\widetilde{T}$
    continue if Prot$_{\mathsf{BC}}$.Status($\widetilde{T}$) is not committed
    get ts$_{\mathsf{closed}}$ := Prot$_{\mathsf{NSB}}$.BlockHeight()
    compute $C^{\mathcal{T}}_{\mathsf{closed}}$ := Cert([$\widetilde{T}$, closed, sid, $\mathcal{T}$, ts$_{\mathsf{closed}}$], $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$)
    call Prot$_{\mathsf{CLI}}$.CloseTrans($C^{\mathcal{T}}_{\mathsf{closed}}$) to negotiate the closed attestation
    call Prot$_{\mathsf{BC}}$.MerkleProof($\widetilde{T}$) to obtain a finalization proof for $\widetilde{T}$
    denote the finalization proof as Merk$^{c_1}_{\mathcal{T}}$ (Figure 7)
    update $\mathcal{S}_{\mathsf{Cert}}$.Add($C^{\mathcal{T}}_{\mathsf{closed}}$) and $\mathcal{S}_{\mathsf{Merk}}$.Add(Merk$^{c_1}_{\mathcal{T}}$)
Daemon Watching(sid, Prot$_{\mathsf{NSB}}$) private:
  ($\mathcal{G}_{\mathcal{T}}$, _, $\mathcal{S}_{\mathsf{Cert}}$, $\mathcal{S}_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  watch four types of attestations {Cert$^{\mathsf{id}}$, Cert$^{o}$, Cert$^{\mathsf{od}}$, Cert$^{c}$}
  process *fresh* attestations via corresponding handlers (see below)
  # Retrieve alternative attestations if necessary.
  for each $\mathcal{T} \in \mathcal{G}_{\mathcal{T}}$ :
    if $\mathcal{T}$.state = opened and Merk$^{c_1}_{\mathcal{T}} \in \mathcal{S}_{\mathsf{Merk}}$ :
      retrieve the roots [$R$, ...] of the proof Merk$^{c_1}_{\mathcal{T}}$
      call Prot$_{\mathsf{NSB}}$.MerkleProof([$R$, ...]) to obtain a status proof Merk$^{c_2}_{\mathcal{T}}$
      continue if Merk$^{c_2}_{\mathcal{T}}$ is not available yet on Prot$_{\mathsf{NSB}}$
      compute the complete proof Merk$^{c}_{\mathcal{T}}$ := [Merk$^{c_1}_{\mathcal{T}}$, Merk$^{c_2}_{\mathcal{T}}$]
      update $\mathcal{T}$.state := closed and $\mathcal{S}_{\mathsf{Merk}}$.Add(Merk$^{c}_{\mathcal{T}}$)
  compute eligible transaction set $\mathcal{S}$ using the current state of $\mathcal{G}_{\mathcal{T}}$
  for each $\mathcal{T} \in \mathcal{S}$:
    continue if $\mathcal{T}$.state is not unknown
    if $\mathcal{T}$.from = Prot$_{\mathsf{CLI}}$:
      compute Cert$^{i}_{\mathcal{T}}$ := Cert([$\mathcal{T}$, init, sid]; $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$)
      call Prot$_{\mathsf{CLI}}$.InitTrans(Cert$^{i}_{\mathcal{T}}$) to request initialization
      call Prot$_{\mathsf{NSB}}$.AddAction(Cert$^{i}_{\mathcal{T}}$) to prove Cert$^{i}_{\mathcal{T}}$ is sent
      update $\mathcal{S}_{\mathsf{Cert}}$.Add(Cert$^{i}_{\mathcal{T}}$) and $\mathcal{T}$.state := init
      non-blocking wait until Prot$_{\mathsf{NSB}}$.MerkleProof(Cert$^{i}_{\mathcal{T}}$) rt. Merk$^{i}_{\mathcal{T}}$
      update $\mathcal{S}_{\mathsf{Merk}}$.Add(Merk$^{i}_{\mathcal{T}}$)
    else: call *self*.SInitedTrans(sid, $\mathcal{T}$)
Upon Receive SInitedTrans(sid, $\mathcal{T}$) private:                    *Northbound*
  ($\mathcal{G}_{\mathcal{T}}$, _, $\mathcal{S}_{\mathsf{Cert}}$, $\mathcal{S}_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  compute and sign the on-chain counterpart $\widetilde{T}$ for $\mathcal{T}$
  compute Cert$^{\mathsf{id}}_{\mathcal{T}}$ := Cert([$\widetilde{T}$, inited, sid, $\mathcal{T}$]; $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$)
  call Prot$_{\mathsf{CLI}}$.InitedTrans(Cert$^{\mathsf{id}}_{\mathcal{T}}$) to request opening of initialized $\mathcal{T}$
  call Prot$_{\mathsf{NSB}}$.AddAction(Cert$^{\mathsf{id}}_{\mathcal{T}}$) to prove Cert$^{\mathsf{id}}_{\mathcal{T}}$ is sent
  update $\mathcal{S}_{\mathsf{Cert}}$.Add(Cert$^{\mathsf{id}}_{\mathcal{T}}$) and $\mathcal{T}$.state := inited
  non-blocking wait until Prot$_{\mathsf{NSB}}$.MerkleProof(Cert$^{\mathsf{id}}_{\mathcal{T}}$) returns Merk$^{\mathsf{id}}_{\mathcal{T}}$
  update $\mathcal{S}_{\mathsf{Merk}}$.Add(Merk$^{\mathsf{id}}_{\mathcal{T}}$)
Upon Receive RInitedTrans(Cert$^{\mathsf{id}}_{\mathcal{T}}$) public:                    *Southbound*
  assert Cert$^{\mathsf{id}}_{\mathcal{T}}$ has the valid form of Cert([$\widetilde{T}$, inited, sid, $\mathcal{T}$]; $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$)
  (_, _, $\mathcal{S}_{\mathsf{Cert}}$, $\mathcal{S}_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  abort if the Cert$^{i}_{\mathcal{T}}$ corresponding to Cert$^{\mathsf{id}}_{\mathcal{T}}$ is not in $\mathcal{S}_{\mathsf{Cert}}$
  assert $\widetilde{T}$ is correctly associated with the wrapper $\mathcal{T}$
  get ts$_{\mathsf{open}}$ := Prot$_{\mathsf{NSB}}$.BlockHeight()
  compute Cert$^{o}_{\mathcal{T}}$ := Cert([$\widetilde{T}$, open, sid, $\mathcal{T}$, ts$_{\mathsf{open}}$]; $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$)
  call Prot$_{\mathsf{CLI}}$.OpenTrans(Cert$^{o}_{\mathcal{T}}$) to request opening for $\mathcal{T}$
  call Prot$_{\mathsf{NSB}}$.AddAction(Cert$^{o}_{\mathcal{T}}$) to prove Cert$^{o}_{\mathcal{T}}$ is sent
  update $\mathcal{S}_{\mathsf{Cert}}$.Add(Cert$^{o}_{\mathcal{T}}$) and $\mathcal{T}$.state := open
  non-blocking wait until Prot$_{\mathsf{NSB}}$.MerkleProof(Cert$^{o}_{\mathcal{T}}$) returns Merk$^{o}_{\mathcal{T}}$
  update $\mathcal{S}_{\mathsf{Merk}}$.Add(Merk$^{o}_{\mathcal{T}}$)
Upon Receive OpenTrans(Cert$^{o}_{\mathcal{T}}$) public:                    *Northbound*
  assert Cert$^{o}_{\mathcal{T}}$ has valid form of Cert([$\widetilde{T}$, open, sid, $\mathcal{T}$, ts$_{\mathsf{open}}$]; $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$)
  (_, _, $\mathcal{S}_{\mathsf{Cert}}$, $\mathcal{S}_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  abort if the Cert$^{\mathsf{id}}_{\mathcal{T}}$ corresponding to Cert$^{o}_{\mathcal{T}}$ is not in $\mathcal{S}_{\mathsf{Cert}}$
  assert ts$_{\mathsf{open}}$ is within a bounded range with Prot$_{\mathsf{NSB}}$.BlockHeight()
  compute Cert$^{\mathsf{od}}_{\mathcal{T}}$ := Cert([$\widetilde{T}$, open, sid, $\mathcal{T}$, ts$_{\mathsf{open}}$]; $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$)
  call Prot$_{\mathsf{BC}}$.Exec($\widetilde{T}$) to trigger on-chain execution
  call Prot$_{\mathsf{CLI}}$.OpenedTrans(Cert$^{\mathsf{od}}_{\mathcal{T}}$) to acknowledge request
  call Prot$_{\mathsf{NSB}}$.AddAction(Cert$^{\mathsf{od}}_{\mathcal{T}}$) to prove Cert$^{\mathsf{od}}_{\mathcal{T}}$ is sent
  update $\mathcal{S}_{\mathsf{Cert}}$.Add(Cert$^{\mathsf{od}}_{\mathcal{T}}$) and $\mathcal{T}$.state := opened
  non-blocking wait until Prot$_{\mathsf{NSB}}$.MerkleProof(Cert$^{\mathsf{od}}_{\mathcal{T}}$) returns Merk$^{\mathsf{od}}_{\mathcal{T}}$
  update $\mathcal{S}_{\mathsf{Merk}}$.Add(Merk$^{\mathsf{od}}_{\mathcal{T}}$)
Upon Receive OpenedTrans(Cert$^{\mathsf{od}}_{\mathcal{T}}$) public:                    *Southbound*
  ast. Cert$^{\mathsf{od}}_{\mathcal{T}}$ has valid form of Cert([$\widetilde{T}$, open, sid, $\mathcal{T}$, ts$_{\mathsf{open}}$]; $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$)
  (_, _, $\mathcal{S}_{\mathsf{Cert}}$, _) := Data[sid]; abort if not found
  abort if the Cert$^{o}_{\mathcal{T}}$ corresponding to Cert$^{\mathsf{od}}_{\mathcal{T}}$ is not in $\mathcal{S}_{\mathsf{Cert}}$
  update $\mathcal{S}_{\mathsf{Cert}}$.Add(Cert$^{\mathsf{od}}_{\mathcal{T}}$) and $\mathcal{T}$.state := 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, $\mathcal{T}$, ts$_{\mathsf{closed}}$], $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$)
  assert $\widetilde{T}$ is finalized on its destination blockchain and obtain Merk$^{c_1}_{\mathcal{T}}$
  assert ts$_{\mathsf{closed}}$ is within a bounded margin with Prot$_{\mathsf{NSB}}$.BlockHeight()
  (_, _, $\mathcal{S}_{\mathsf{Cert}}$, $\mathcal{S}_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  compute Cert$^{c}_{\mathcal{T}}$ := Cert([$\widetilde{T}$, closed, sid, $\mathcal{T}$, ts$_{\mathsf{closed}}$], $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$)
  call Prot$_{\mathsf{CLI}}$.ClosedTrans(Cert$^{c}_{\mathcal{T}}$) to acknowledged request
  update $\mathcal{S}_{\mathsf{Cert}}$.Add(Cert$^{c}_{\mathcal{T}}$), $\mathcal{S}_{\mathsf{Merk}}$.Add(Merk$^{c_1}_{\mathcal{T}}$) and $\mathcal{T}$.state := closed
Upon Receive ClosedTrans(Cert$^{c}_{\mathcal{T}}$) public:                    *Bidirectional*
  ast. Cert$^{c}_{\mathcal{T}}$ has valid form of Cert([$\widetilde{T}$, closed, sid, $\mathcal{T}$, ts$_{\mathsf{closed}}$], $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{D}}_{\mathsf{sid}}$)
  (_, _, $\mathcal{S}_{\mathsf{Cert}}$, _) := Data[sid]; abort if not found
  abort if Cert([$\widetilde{T}$, closed, sid, $\mathcal{T}$, ts$_{\mathsf{closed}}$], $\mathsf{Sig}^{\mathcal{V}}_{\mathsf{sid}}$) is not in $\mathcal{S}_{\mathsf{Cert}}$
  update $\mathcal{S}_{\mathsf{Cert}}$.Add(Cert$^{c}_{\mathcal{T}}$) and $\mathcal{T}$.state := closed
Daemon Redeem(sid) private:
  # Invoke the insurance contract periodically
  ($\mathcal{G}_{\mathcal{T}}$, cid, $\mathcal{S}_{\mathsf{Cert}}$, $\mathcal{S}_{\mathsf{Merk}}$) := Data[sid]; abort if not found
  for each *unclaimed* $\mathcal{T} \in \mathcal{G}_{\mathcal{T}}$:
    get the Cert$_{\mathcal{T}}$ from $\mathcal{S}_{\mathsf{Cert}} \bigcup \mathcal{S}_{\mathsf{Merk}}$ with the most advanced state
    call Prot$_{\mathsf{ISC}}$.InsuranceClaim(cid, Cert$_{\mathcal{T}}$) to claim insurance
```

Figure 6: Protocol description of of Prot$_{\mathsf{VES}}$. Gray background denotes non-blocking operations triggered by status updates on Prot$_{\mathsf{NSB}}$. Handlers annotated with *northbound* and *southbound* process transactions originated from Prot$_{\mathsf{VES}}$ and Prot$_{\mathsf{CLI}}$, respectively. Handlers annotated with *bidirectional* are shared by all transactions.

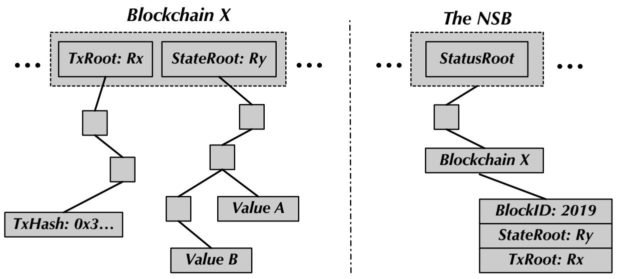

Figure 7: The complete on-chain proof (denoted by $\text{Merk}^{c}_{\mathcal{T}}$) to prove that the state of a transaction is eligible to be promoted as closed. The left-side part is the finalization proof (denoted by $\text{Merk}^{c_1}_{\mathcal{T}}$) for the transaction collected from its destination blockchain; the right-side part is the blockchain status proof (denoted by $\text{Merk}^{c_2}_{\mathcal{T}}$) collected from the NSB.

the $\text{Cert}^{o}_{\mathcal{T}}$ received from Prot_{CLI} and a post-open (*i.e.,* opened) certificate $\text{Cert}^{od}_{\mathcal{T}}$ computed by Prot_{VES} is that latter one is signed by both parties. Only the ts_{open} specified in $\text{Cert}^{od}_{\mathcal{T}}$ is used by Prot_{ISC} when evaluating the deadline constraint of $\mathcal{T}$.

Southbound transactions originating from Prot_{CLI} are processed by Prot_{VES} in a similar manner as the northbound transactions, via the RInitedTrans and OpenedTrans handlers. We clarify a subtlety in the RInitedTrans handler when verifying the *association* between $\widetilde{T}$ and $\mathcal{T}$ (line 61). If $\widetilde{T}$ depends on the resulting state from its upstream transactions (for instance, T4 depends on the resulting state of T1 in Figure 4), Prot_{VES} needs to verify that the state used by $\widetilde{T}$ is consistent with the state enclosed in the finalization proofs of those upstream transactions.

4.2.3 Proactive Watching Services

The cross-chain execution process proceeds when all session-relevant blockchains and the NSB make progresses. As the driver of execution, Prot_{VES} internally creates two watching services to *proactively* read the status of those blockchains.

In the watching daemon to one blockchain, Prot_{VES} mainly reads the public ledger of Prot_{BC} to monitor the status of transactions that have been posted for on-chain execution. If Prot_{VES} notices that an on-chain transaction $\widetilde{T}$ is recently finalized, it requests the closing process for $\mathcal{T}$ by sending Prot_{CLI} a timestamped certificate C_{closed}. The pair of handlers, CloseTrans and ClosedTrans, are used by both Prot_{VES} and Prot_{CLI} in this exchange. Both handlers can be used for handling northbound and southbound transactions, depending on which party sends the closing request. In general, a transaction's originator has a stronger motivation to initiate the closing process because the originator would be held accountable if the transaction were not timely closed by its deadline.

In addition, Prot_{VES} needs to retrieve a Merkle Proof from Prot_{BC} to prove the finalization of $\widetilde{T}$. This proof, denoted by $\text{Merk}^{c_1}_{\mathcal{T}}$, serves two purposes: (i) it is the first part of a complete on-chain proof to prove that the state $\widetilde{T}$ can be promoted to closed, as shown in Figure 7; (ii) if the resulting state of $\widetilde{T}$ is used by its downstream transactions, $\text{Merk}^{c_1}_{\mathcal{T}}$ is necessary to ensure that those downstream transactions indeed use genuine state.

In the watching service to Prot_{NSB}, Prot_{VES} performs following tasks. First, as described in § 4.1.2, NSB is treated as a fallback communication medium for the off-chain channel. Thus, Prot_{VES} searches the sorted ActionMT to look for any session-relevant certificates that have not been received via the off-chain channel. Second, for each opened $\mathcal{T}$ whose closed attestation is still missing after Prot_{VES} has sent C_{closed} (indicating slow or no reaction from Prot_{CLI}), Prot_{VES} tries to retrieve the second part of $\text{Merk}^{c}_{\mathcal{T}}$ from Prot_{NSB}. The second proof, denoted as $\text{Merk}^{c_2}_{\mathcal{T}}$, is to prove that the Merkle roots referred in $\text{Merk}^{c_1}_{\mathcal{T}}$ are correctly linked to a StatusRoot on a finalized NSB block (see Figure 7). Once $\text{Merk}^{c}_{\mathcal{T}}$ is fully constructed, the state of $\mathcal{T}$ is promoted as closed. Finally, Prot_{VES} may find a new set of transactions that are eligible to be executed if their preconditions are finalized due to any recently-closed transactions. If so, Prot_{VES} processes them by either requesting initialization from Prot_{CLI} or calling SInitedTrans internally, depending on the originators of those transactions.

4.2.4 Prot_{ISC} Invocation

Prot_{VES} periodically invokes Prot_{ISC} to execute the contract. All internally stored certificates and *complete* Merkle proofs are acceptable. However, for any $\mathcal{T}$, Prot_{VES} should invoke Prot_{ISC} only using the attestation with the most advanced state, since lower-ranked attestations for $\mathcal{T}$ are effectively ignored by Prot_{ISC} (*c.f.,* § 4.4).

4.3 Execution Protocol by dApp Clients

Prot_{CLI} specifies the protocol implemented by dApp clients. Prot_{CLI} defines the following set of handlers to match Prot_{VES}. In particular, the InitedTrans and OpenedTrans match the SInitedTrans and OpenTrans of Prot_{VES}, respectively, to process Cert^{id} and Cert^{od} sent by Prot_{VES} when handling transactions originated from Prot_{VES}. The InitTrans and OpenTrans process Cert^{i} and Cert^{o} sent by Prot_{VES} when executing transactions originated from Prot_{CLI}. The CloseTrans and ClosedTrans of Prot_{CLI} match their counterparts in Prot_{VES} to negotiate closing attestations.

For usability, HyperService imposes smaller requirements on the watching daemons implemented by Prot_{CLI}. Specially, Prot_{CLI} still proactively watches Prot_{NSB} to have a fallback communication medium with Prot_{VES}. However, Prot_{CLI} is *not* required to proactively watch the status of underlying blockchains or dynamically compute eligible transactions whenever the execution status changes. We intentionally offload such complexity on Prot_{VES} to enable lightweight dApp clients. Prot_{CLI}, though, should (and is motivated to) check the status of self-originated transactions in order to request transaction closing.

4.4 Protocol Realization of the ISC

Figure 8 specifies the protocol realization of the ISC. The CreateContract handler is the entry point of requesting insurance contract creation using Prot_{ISC}. It generates the arbitration code, denoted as *contract*, based on the given dApp executable $\mathcal{G}_{\mathcal{T}}$. The *contract* internally uses T_{state} to track the state of each transaction in $\mathcal{G}_{\mathcal{T}}$, which is updated when processing security attestations in the InsuranceClaim handler. For clear presentation, Figure 8 extracts the state proof and fund reversion tuple from $\mathcal{T}$ as dedicated variables st_{proof} and A_{revs}. When the Prot_{ISC} times out, it executes the contract terms based on its internal state, after which its funds

```
Init: Data := $\emptyset$
Upon Receive CreateContract($\mathcal{G}_{\mathcal{T}}$):
  generate the arbitration cod, denoted by *contract*, as follows
  initialize three maps $T_{state}$, $A_{revs}$ and $F_{stake}$
  for each $\mathcal{T} \in \mathcal{G}_{\mathcal{T}}$ :
    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, $st_{proof}$]
    retrieve tid's fund-reversion account, denoted as dst
    initialize $A_{revs}$[tid] := [amt=0, dst]
  compute an identifier for *contract* as cid := $H(\overrightarrow{0}$, *contract*)
  initialize Data[cid] := [$\mathcal{G}_{\mathcal{T}}$, $T_{state}$, $A_{revs}$, $F_{stake}$]
  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.state is *more advanced* the state enclosed by Cert
  if Atte is a certificate signed by both parties :
    assert SigVerify(Atte) is true
    if Atte is $Cert^{od}_{\mathcal{T}}$ : update T.state := opened; T.$ts_{open}$ := Atte.$ts_{open}$
    else : update T.state := 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 $Merk^{i}_{\mathcal{T}}$ or $Merk^{id}_{\mathcal{T}}$ or $Merk^{o}_{\mathcal{T}}$ :
      retrieve the certificate $Cert^{i}_{\mathcal{T}}$ or $Cert^{id}_{\mathcal{T}}$ or $Cert^{o}_{\mathcal{T}}$ from Atte
      assert the $\widetilde{T}$ enclosed in $Cert^{id}_{\mathcal{T}}$ or $Cert^{o}_{\mathcal{T}}$ is genuine
      assert the $ts_{open}$ enclosed in $Cert^{o}_{\mathcal{T}}$ is genuine
      update T.state := Atte.state
    elif Atte is $Merk^{od}_{\mathcal{T}}$ :
      retrieve the certificate $Cert^{od}_{\mathcal{T}}$ from Atte
      update T.state := opened and T.$ts_{open}$ := $Cert^{od}_{\mathcal{T}}$.$ts_{open}$
    elif Atte is $Merk^{c}_{\mathcal{T}}$ :
      update T.$st_{proof}$ based on $Merk^{c_1}_{\mathcal{T}}$ if necessary
      update T.$ts_{closed}$ as the height of the block attaching $Merk^{c_2}_{\mathcal{T}}$
      update T.state := closed
Upon Timeout SettleContract(cid):                                  Internal Daemon
  ($\mathcal{G}_{\mathcal{T}}$, $T_{state}$, $A_{revs}$, $F_{stake}$) := Data[cid]; abort if not found
  for (tid, T) $\in$ $T_{state}$ :
    continue if T.state is not closed
    update $A_{revs}$[tid].amt := T.$\mathcal{T}$.meta.amt
    if DeadlineVerify(T) = true : update T.state := correct
  compute $\mathcal{S}$ := DirtyTrans($\mathcal{G}_{\mathcal{T}}$, $T_{state}$) # non-empty if execution 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.state = closed | open | opened : *resp*[tid] := T.$\mathcal{T}$.from
    elif T.state = inited : *resp*[tid] := T.$\mathcal{T}$.to
    elif T.state = init : *resp*[tid] := $\mathcal{D}$
    else : *resp*[tid] := $\mathcal{V}$
  return any remaining funds in $F_{stake}$ to corresponding senders
  call Data.erase[cid] to stay silent afterwards
```

Figure 8: Prot$_{ISC}$: the protocol realization of the ISC arbitrator.

are depleted and the contract never runs again. Below we explain several technical subtleties.

4.4.1 *Insurance Claim*

The InsuranceClaim handler processes security attestations from Prot$_{VES}$ and Prot$_{CLI}$. Only dual-signed certificates (*i.e.,* Cert^{od} and Cert^{c}) or complete Merkle proofs are acceptable. Processing dual-signed certificates is straightforward as they are explicitly agreed by both parties. However, processing Merkle proof requires additional correctness checks. First, when validating a Merkle proof $Merk^{i}_{\mathcal{T}}$, $Merk^{id}_{\mathcal{T}}$ or $Merk^{o}_{\mathcal{T}}$, Prot$_{ISC}$ retrieves the single-party signed certificate $Cert^{i}_{\mathcal{T}}$, $Cert^{id}_{\mathcal{T}}$ or $Cert^{o}_{\mathcal{T}}$ 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.,* $Cert^{i}_{\mathcal{T}}$ is signed by Prot$_{VES}$, $Cert^{id}_{\mathcal{T}}$ is signed by $\mathcal{T}$'s originator and $Cert^{o}_{\mathcal{T}}$ is signed by the destination of $\mathcal{T}$. (ii) The enclosed on-chain transaction $\widetilde{T}$ in $Cert^{id}_{\mathcal{T}}$ and $Cert^{o}_{\mathcal{T}}$ is correctly associated with $\mathcal{T}$. The checking logic is the same as the on used by Prot$_{VES}$, which has been explained in § 4.2.2. (iii) The enclosed ts_{open} in $Cert^{o}_{\mathcal{T}}$ is genuine, where the genuineness is defined as a bounded difference between ts_{open} and the height of the NSB block that attaches $Merk^{o}_{\mathcal{T}}$.

4.4.2 *Contract Term Settlement*

Prot$_{ISC}$ registers a callback SettleContract to execute contract terms automatically upon timeout. Prot$_{ISC}$ internally defines an additional

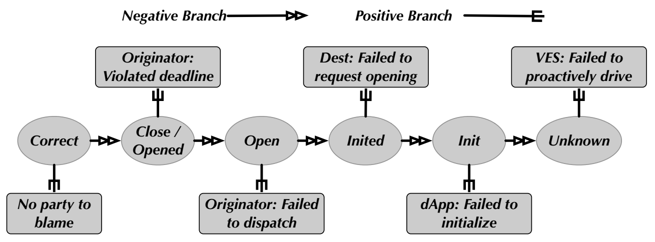
Negative Branch
Positive Branch
Originator: Violated deadline
Dest: Failed to request opening
VES: Failed to proactively drive
Correct
Close / Opened
Open
Inited
Init
Unknown
No party to blame
Originator: Failed to dispatch
dApp: Failed to initialize

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

transaction state, called correct. The state of a closed transaction is promoted to correct if its deadline constraint is satisfied. Then, Prot$_{ISC}$ computes the possible *dirty* transactions in $\mathcal{G}_{\mathcal{T}}$, which are the transactions that are eligible to be opened, but with non-correct state. Thus, the execution succeeds only if $\mathcal{G}_{\mathcal{T}}$ has no dirty transactions. Otherwise, Prot$_{ISC}$ employs a decision tree, shown in Figure 9, to decide the responsible party for each dirty transaction. The decision tree is derived from the execution steps taken by Prot$_{VES}$ and Prot$_{CLI}$. In particular, if a transaction $\mathcal{T}$'s state 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 state is inited, then it is $\mathcal{T}$'s destination party's responsibility for not proceeding with $\mathcal{T}$ even though $Cert^{id}_{\mathcal{T}}$ has been provably sent. If a transaction $\mathcal{T}$'s state 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 $\mathsf{Cert}^i_{\mathcal{T}}$ sent by $\mathcal{V}$. Finally, if transaction $\mathcal{T}$'s state is unknown, then $\mathcal{V}$ is held accountable for not proactively driving the initialization of $\mathcal{T}$, no matter which party originates $\mathcal{T}$.

4.5 $\mathsf{Prot}_{\mathsf{BC}}$ and $\mathsf{Prot}_{\mathsf{NSB}}$

$\mathsf{Prot}_{\mathsf{BC}}$ specifies the protocol realization of a general-purpose blockchain where a set of consensus nodes run a secure protocol to agree upon the public global state. In this paper, we regard $\mathsf{Prot}_{\mathsf{BC}}$ as a conceptual party trusted for correctness and availability, *i.e.,* $\mathsf{Prot}_{\mathsf{BC}}$ 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. $\mathsf{Prot}_{\mathsf{NSB}}$ specifies the protocol realization of the NSB. $\mathsf{Prot}_{\mathsf{NSB}}$ is an extended version of $\mathsf{Prot}_{\mathsf{BC}}$ with additional capabilities. Due to space constraint, we defer the detailed protocol description of $\mathsf{Prot}_{\mathsf{BC}}$ and $\mathsf{Prot}_{\mathsf{NSB}}$ in Appendix A.1.

4.6 Main Security Theory

To rigorously prove the security properties of UIP, we first present the cryptography abstraction of the UIP in form of an ideal functionality $\mathcal{F}_{\mathsf{UIP}}$. The ideal functionality articulates the correctness and security properties that UIP wishes to attain by assuming a trusted entity. Then we prove that $\mathsf{Prot}_{\mathsf{UIP}}$, our the decentralized real-world protocol, securely realizes $\mathcal{F}_{\mathsf{UIP}}$ using the UC framework [25], *i.e.,* $\mathsf{Prot}_{\mathsf{UIP}}$ achieves the same functionality and security properties as $\mathcal{F}_{\mathsf{UIP}}$ without assuming any trusted authorities. Since the rigorous proof requires non-trivial simulator construction within the UC framework, we defer those details to Appendix A.2.

5 IMPLEMENTATION AND EXPERIMENTS

In this section, we present the implementation of a HyperService prototype and report some experiment results on the prototype. The total development effort includes (i) ~1,500 lines of Java code and ~3,100 lines of ANTLR [53] grammar code for building the HSL programming framework, (ii) ~9,500 lines of code, in both Go and Python, for building UIP (including the NSB) and (iii) ~1,000 lines of code, in Solidity, Vyper, Go and HSL, for writing cross-chain dApps running on HyperService. Source code is available at [17].

5.1 Platform Implementation

To demonstrate the interoperability and programmability across heterogeneous blockchains on HyperService, our current prototype incorporates Ethereum, the flagship public blockchain, and a permissioned blockchain built atop the Tendermint [14] consensus engine, a commonly cited corestone for building enterprise blockchains. We implement the necessary accounts (wallets), smart contract environment, and on-chain storage to make the permissioned blockchain with full programmability. The NSB is also built atop Tendermint with full support for its claimed capabilities, such as action staking and Merkle proof retrieval.

For the programming framework, we implement the HSL compiler that takes HSL programs and contracts written in Solidity, Vyper, and Go as input, and produces transaction dependency graphs. We implement the multi-lang and the HSL front ends of the compiler using ANTLR [53], which parse the input HSL program and contracts, build the intermediate representation of the HSL program, and convert the types of contract entities into our unified types. We also implement the validation component that analyzes the intermediate representation of the HSL program to validate the entities, operations, and dependencies specified in the HSL program.

Our experience with the prototype implementation is that *the effort for horizontally scaling HyperService to incorporate a new blockchain is lightweight*, requiring zero protocol change to both UIP and the blockchain itself. We simply need to add an extra parser to the HSL front end to support the programming language used by the blockchain (if this language has not been supported), and meanwhile VESes extends their visibility to this blockchain. At the time of this writing, we are working on extending HyperService to include Nebulas [9] and blockchains built atop Substrate [13], another blockchain-building SDK similar to Tendermint.

5.2 Application Implementation

Besides the platform implementation, we further implement and deploy three categories of cross-chain dApps on HyperService.

Financial Derivatives. Financial derivatives are among the mostly cited blockchain applications. However, external data feed, *i.e.,* an oracle, is often required for financial instructions. Currently, oracles are either built atop trusted third-party providers (*e.g.,* Oraclize [10]), or using trusted hardware enclaves [60]. HyperService, for the first time, realizes the possibility of *using blockchains themselves as oracles*. With the built-in decentralization and correctness guarantee of blockchains, HyperService fully avoids trusted parties while delivering genuine data feed to smart contracts. In this application sector, we implement a cross-chain cash-settled Option dApp in which options can be simultaneously tradeable on different blockchains (a scaled-up version of the introductory example discussed in § 2.3).

Cross-chain Asset Movement. HyperService natively enables cross-chain asset transfers without relying on any trusted entities, such as exchanges. This primitive could power a wide range of applications, such as a global payment network that interconnects geographically distributed bank-backed consortium blockchains [8], an initial coin offering in which tokens can be sold in various cryptocurrency, and a gaming platform where players can freely trade and redeem their valuables across different games. In this category, we implement an asset movement dApp with hybrid operations where assets are moved among accounts and smart contracts across different blockchains

Federated Computing. In a federated computing model, all participants collectively work on an umbrella task by submitting their local computation results. In the scenario where transparency and accountability are desired, blockchains are perfect platforms for persisting both the results submitted by each participant and the logic for aggregating those results. In this application category, we implement a federated voting system where delegates in different region can submit their votes to their regional blockchains, and the logic for computing the final vote based on the regional votes is publicly visible on another blockchain.

	Financial Derivatives		CryptoAsset Movement		Federated Computing	
	Mean	%	Mean	%	Mean	%
HSL Compilation	1.1769	∼16	0.2598	∼4	1.095	∼15
ISC Initialization	4.2399	∼58	4.1529	∼67	4.2058	∼60
Action/Status Staking	0.6754	∼10	0.7295	∼12	0.7592	∼11
Proof Retrieval	1.0472	∼15	1.0511	∼17	0.9875	∼14
Total	7.1104		6.1933		7.0475	

Table 3: End-to-end dApp execution latency on HyperService, with profiling breakdown. All times are in seconds.

5.3 Experiments

We ran experiments with three blockchain testnets: one private Ethereum testnet, one Tendermint-based blockchain, and the NSB. Each of those testnets is deployed on a VM instance of a public cloud on different continents. For experiment purpose, dApp clients and VES nodes can be deployed either locally or on cloud.

5.3.1 End-to-End Latency

We evaluated all three applications mentioned in § 5.2 and reported their end-to-end execution latency on HyperService in Table 3. The reported latency includes HSL program compiling, dApp-VES session creation, and (batched) NSB action staking and proof retrieval during UIP protocol exchange. All reported times include the networking latency across the global Internet. Each datapoint is the average of more than one hundred runs. We do not include the latency for actual on-chain execution since the consensus efficiency of different blockchains varies and is not controlled by HyperService. We also do not include the time for ISC insurance claim in the e2e latency because insurance claim can be done offline anytime before the ISC expiration.

These dApps show similar latency profiling breakdown, where the session setup is the most time consuming phase because it requires handshakes between the dApp client and VES, and also includes the time for ISC deployment and initialization. The CryptoAsset dApp has much lower HSL compilation latency since its operation only involves one smart contract, whereas the rest two dApps import three contracts written in Go, Vyper and Solidity. In each dApp, all its NSB related operations (*e.g.,* action/status staking and proof retrievals) are bundled and performed in a batch for experiment purpose, even though all certificates required for ISC arbitration have been received via off-chain channels. The sizes of actions and proofs for three dApps are difference since their executables contain different number of transactions.

5.3.2 NSB Throughput and HyperService Capacity

The throughout of the NSB affects aggregated capacity of HyperService. As discussed in § 4.1.3, the NSB is horizontally shardable at the granularity of each underlying blockchain. Our current NSB prototype is an non-sharded version with about 10 thousand transactions per second. From dApp's perspective, each transaction in its executable maximally spawns six NSB-transactions (five action staking and one status staking), assuming that the off-chain channel is fully nonfunctional, and minimally zero NSB-transaction. Thus, the *lower bound* of the aggregated dApp throughput on HyperService, which would happen only if all off-chain channels among dApp clients and VESes ware broken, is about ∼1700 transactions per second, which is already three orders of magnitude faster than the speed of Ethereum mainnet [7]. To future-proof NSB for faster blockchains, we could bundle several action or status staking made by different dApps into a single NSB-transaction, so that the effective throughput perceived by dApps may go beyond the speed limit of the NSB itself. We leave such NSB related optimization to active future work.

6 RELATED WORK AND DISCUSSIONS

Blockchain interoperability is often considered as one of the prerequisites for the massive adoption of blockchains. The recent academic proposals have mostly focused on moving tokens between two blockchains via trustless exchange protocol, including side-chains [20, 32, 37], atomic cross-chain swaps [1, 34], and cryptocurrency-backed assets [59]. However, programmability is largely ignored in these protocols.

In industry, Cosmos [4] and Polkadot [11] are two notable projects that advocate blockchain interoperability. They share the similar spirit: each of them has a consensus engine to build blockchains (*i.e.,* Tendermint [14] for Cosmos and Substrate [13] for Polkadot), and a *mainchain* (*i.e.,* the Hub in Cosmos and RelayChain for Polkadot) to bridge individual blockchains. Although we do share the similar vision of "an Internet of blockchains", there are two notable differences between them and HyperService. First and foremost, the recent development of Cosmos is heading towards *homogeneity* where only Tendermint-powered blockchains are interoperable [5]. This is in fundamental contrast with HyperService where the blockchain heterogeneity is a first-class design requirement. Polkadot proceeds relatively slower: Substrate is still in early stage [13]. In addition, neither of them has a unified programming framework for writing cross-chain dApps.

Existing blockchain platforms such as Ethereum [57] and Nebulas [9] allow developers to write contracts using new languages such as Solidity [12] and Vyper [16] or the tailored version of the existing languages such as Go, Javascript, and C++. To unify these heterogeneous programming languages, HyperService invents HSL which has a multi-lang front end to parse those contacts and convert their types to unified types. Although there exist domain-specific languages in a variety of network security fields that have a well-established corpus of low level algorithms, such as secure overlay networks [38, 48] and network intrusions [21, 55, 56], these languages are explicitly designed to solve their domain-specific problems, and cannot meet the needs of unified programming framework for writing cross-chain dApps.

7 CONCLUSION

In this paper, we presented HyperService, the first platform that offers interoperability and programmability across heterogeneous blockchains. HyperService is powered by innovative designs: HSL, a programming framework for writing cross-chain dApps by unifying smart contracts written in different languages, and UIP, the universal blockchain interoperability protocol designed to securely realize the complex operations defined in these dApps on blockchains. We implemented a HyperService prototype in about 14,000 lines of code to demonstrate its practicality, and ran experiments on the prototype to report the end-to-end execution latency for dApps, as well as the aggregated platform throughput.

REFERENCES

[1] Bitcoin Wiki: Atomic Cross-Chain Trading. https://en.bitcoin.it/wiki/Atomic_swap.
[2] CoinMarketCap. https://coinmarketcap.com.
[3] Connect Your Smart Contract with Real-world Data. https://rhombus.network.
[4] Cosmos. https://cosmos.network.
[5] Cosmos WhitePaper. https://cosmos.network/resources/whitepaper.
[6] DPOS Consensus Algorithm. https://steemit.com/dpos/@dantheman/dpos-consensus-algorithm-this-missing-white-paper.
[7] Etherscan: The Ethereum Blockchain Explorer. https://etherscan.io/.
[8] J.P. Morgan: Blockchain and Distributed Ledger. https://www.jpmorgan.com/global/blockchain.
[9] Nebulas. https://github.com/nebulasio.
[10] Oraclize. http://www.oraclize.it.
[11] Polkadot. https://polkadot.network.
[12] Solidity. https://solidity.readthedocs.io/en/v0.5.6/.
[13] Substrate: The platform for blockchain innovators. https://github.com/paritytech/substrate.
[14] Tendermint Core. https://tendermint.com.
[15] Tor Directory Authorities. https://metrics.torproject.org/rs.html#search/flag:authority.
[16] Vyper. https://github.com/ethereum/vyper.
[17] HyperService: Interoperability and Programmability across Heterogeneous Blockchains. https://sites.google.com/view/hyperservice, 2019.
[18] Monoxide: Scale Out Blockchain with Asynchronized Consensus Zones. In *USENIX Symposium on Networked Systems Design and Implementation* (2019).
[19] 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).
[20] Back, A., Corallo, M., Dashjr, L., Friedenbach, M., Maxwell, G., Miller, A., Poelstra, A., Timón, J., and Wuille, P. Enabling Blockchain Innovations With Pegged Sidechains. *URL: http://www. opensciencereview. com/papers/123/enablingblockchain-innovations-with-pegged-sidechains* (2014), 72.
[21] Borders, K., Springer, J., and Burnside, M. Chimera: A declarative language for streaming network traffic analysis. In *USENIX Security* (2012).
[22] 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).
[23] Buterin, V. Chain Interoperability. https://static1.squarespace.com/static/55f73743e4b051cfcc0b02cf/t/5886800ecd0f68de303349b1/1485209617040/Chain+Interoperability.pdfi.
[24] Buterin, V., et al. A Next-Generation Smart Contract and Decentralized Application Platform. *white paper* (2014).
[25] Canetti, R. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *IEEE Symposium on Foundations of Computer Science* (2001).
[26] 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).
[27] Costan, V., and Devadas, S. Intel SGX explained. https://eprint.iacr.org/2016/086.pdf.
[28] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium* (2004).
[29] 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.
[30] 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.
[31] Garay, J., Kiayias, A., and Leonardos, N. The Bitcoin Backbone Protocol with Chains of Variable Difficulty. In *Annual International Cryptology Conference* (2017), Springer, pp. 291–323.
[32] Gazi, P., Kiayias, A., and Zindros, D. Proof-of-stake Sidechains. In *IEEE Symposium on Security & Privacy* (2019).
[33] 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.
[34] Herlihy, M. Atomic Cross-Chain Swaps. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing* (2018), ACM, pp. 245–254.
[35] 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.
[36] Khalil, R., and Gervais, A. Revive: Rebalancing Off-blockchain Payment Networks. In *ACM SIGSAC Conference on Computer and Communications Security* (2017), pp. 439–453.
[37] Kiayias, A., and Zindros, D. Proof-of-work Sidechains. Tech. rep., Cryptology ePrint Archive, Report 2018/1048, 2018.
[38] Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. Mace: Language support for building distributed systems. In *PLDI* (2007).
[39] 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 Architecuture* (2014), pp. 361–372.
[40] 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).
[41] 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).
[42] 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.
[43] 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.
[44] Krupp, J., and Rossow, C. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *27th USENIX Security Symposium* (2018), pp. 1317–1333.
[45] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* (1978).
[46] Lamport, L., Shostak, R., and Pease, M. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1982).
[47] 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).
[48] Loo, B. T., Condie, T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., and Stoica, I. Declarative networking: Language, execution and optimization. In *SIGMOD* (2006).
[49] 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.
[50] 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.
[51] 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.
[52] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008.
[53] Parr, T. Antlr, 2014. https://www.antlr.org/.
[54] 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.
[55] Sommer, R., Vallentin, M., De Carli, L., and Paxson, V. Hilti: An abstract execution environment for deep, stateful network traffic analysis. In *IMC* (2014).
[56] Vallentin, M., Paxson, V., and Sommer, R. Vast: A unified platform for interactive network forensics. In *NSDI* (2016).
[57] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* (2014).
[58] 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.
[59] Zamyatin, A., Harz, D., Lind, J., Panayiotou, P., Gervais, A., and Knottenbelt, W. XCLAIM: Trustless, Interoperable, Cryptocurrency-Backed Assets.
[60] Zhang, F., Cecchetti, E., Croman, K., Juels, A., and Shi, E. Town Crier: An Authenticated Data Feed for Smart Contracts. In *ACM CCS* (2016).
[61] 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.

A APPENDIX

A.1 Specification of $\mathsf{Prot}_{\mathsf{NSB}}$ and $\mathsf{Prot}_{\mathsf{BC}}$

The detailed protocol description of $\mathsf{Prot}_{\mathsf{NSB}}$ and $\mathsf{Prot}_{\mathsf{BC}}$ is given in Figure 10. We model block generation and consensus in $\mathsf{Prot}_{\mathsf{NSB}}$ (and $\mathsf{Prot}_{\mathsf{BC}}$) as a *discrete clock* that proceeds in *epochs*. The length of an epoch is not fixed to reflect the consensus. At the end of each epoch, a new block is packaged and added to the append-only public Ledger.

The block format of $\mathsf{Prot}_{\mathsf{NSB}}$ 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 $\mathsf{Prot}_{\mathsf{NSB}}$ and $\mathsf{Prot}_{\mathsf{BC}}$. StatusMT and ActionMT are constructed using 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 or random selections), and will be rolled over to the next epoch.

StatusPool is constructed via the CloseureClaim interfaces executed by all NSB peers. The CloseureClaim directly listens for transactions claimed by HyperService users (VESes and dApp clients). This design avoids including irrelevant transactions that are not generated by any HyperService sessions into the StatusMT. An alternative design for building StatusPool is via the CloseureWatching which proactively watches all underlying blockchains to collected the transaction Merkle roots and state roots packaged in recently finalized blocks. This design is more cost efficient for HyperService users since they do not need to explicitly claim these roots. Meanwhile, the structure of the StatusMT is changed to the same as ActionMT since all its stored data now become static.

Figure 10 further specifies the protocol of each individual 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 the security parameter $\mathcal{K}$ [46].

The ActionPool is constructed in a similar manner as the StatusPool. In Figure 10, an interface annotated with override is also implemented by $\mathsf{Prot}_{\mathsf{BC}}$, although the implementation detail may be different; for instance $\mathsf{Prot}_{\mathsf{BC}}$ may have different consensus process than $\mathsf{Prot}_{\mathsf{NSB}}$ in the DiscreteTimer interface.

A.2 Formal Proof of Our Main Theorem

In this section, we present the main security theorem of our our cryptography protocol UIP, and rigorously prove it using the UC-framework [25].

A.2.1 *Ideal Functionality $\mathcal{F}_{UIP}$*

We first present the cryptography abstraction of the UIP in form of an *ideal functionality* $\mathcal{F}_{\mathsf{UIP}}$. The ideal functionality articulates the correctness and security properties that HyperService wishes to attain by assuming a trusted entity. The detailed description of $\mathcal{F}_{\mathsf{UIP}}$ is given in Figure 11. Below we provide additional explanations.

Session Setup. Through this interface, a pair of parties $(\mathcal{P}_a, \mathcal{P}_z)$ (*e.g.,* a dApp client and a VES) requests $\mathcal{F}_{\mathsf{UIP}}$ to securely execute a dApp executable. They provide the executable in form of a transaction dependency graph $\mathcal{G}_T$, as well as the correctness arbitration code *contract* As a trusted entity, $\mathcal{F}_{\mathsf{UIP}}$ computes the generates keys for both parties, allowing $\mathcal{F}_{\mathsf{UIP}}$ to sign transactions and compute certificates on their behalf. Both parties are required to stake sufficient funds, derived from the *contract*, into $\mathcal{F}_{\mathsf{UIP}}$. $\mathcal{F}_{\mathsf{UIP}}$ annotates each transaction wrapper $\mathcal{T}$ 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 $\mathcal{F}_{\mathsf{UIP}}$ with the real-world protocol $\mathsf{Prot}_{\mathsf{UIP}}$, in Figure 11, we assume that $\mathcal{P}_a$ is the dApp client and $\mathcal{P}_z$ is the VES.

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

Transaction State Updates. $\mathcal{F}_{\mathsf{UIP}}$ defines a set of interfaces to accept external calls for updating transaction state. In each interface, $\mathcal{F}_{\mathsf{UIP}}$ performs necessary correctness check to guarantee that the state promotion is legitimate. In all interfaces, $\mathcal{F}_{\mathsf{UIP}}$ computes an attestation for the corresponding transaction state, and sends it to both parties to formally notify the actions taken by $\mathcal{F}_{\mathsf{UIP}}$.

Financial Term Execution. Upon the expiration of *timer*, both parties can invoke the TermExecution interface to trigger the code contract execution. The arbitration logic is also derived from decision tree mentioned in Figure 9. However, $\mathcal{F}_{\mathsf{UIP}}$ decides the final state of each transaction merely using its internal state due to the assumed trustiness.

Verbose Definition of $\mathcal{F}_{\mathsf{UIP}}$. We *intentionally* define $\mathcal{F}_{\mathsf{UIP}}$ verbosely (that is, sending many signed messages) in order to accurately match $\mathcal{F}_{\mathsf{UIP}}$ to the real world protocol $\mathsf{Prot}_{\mathsf{UIP}}$. For instance, in the SessionCreate interface, $\mathcal{F}_{\mathsf{UIP}}$ certifies $(\mathcal{G}_T, \mathit{contract}, \mathsf{sid})$ 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 those state update interfaces are not essential to ensure the correctness of execution due to the assumed trustiness of $\mathcal{F}_{\mathsf{UIP}}$. However, $\mathcal{F}_{\mathsf{UIP}}$ still publishes attestations to emulate the *side effects* of $\mathsf{Prot}_{\mathsf{UIP}}$ in the real world. As we shall see, the mulation is crucial to prove that $\mathcal{F}_{\mathsf{UIP}}$ UC-realizes $\mathsf{Prot}_{\mathsf{UIP}}$.

A.2.2 *Correctness and Security Properties of $\mathcal{F}_{UIP}$*

With the assumed trustiness, it is not hard to see that $\mathcal{F}_{\mathsf{UIP}}$ offers the following correctness and security properties. First, after the pre-agreed timeout, the execution either finishes correctly with all precondition and deadline rules satisfied, or the execution fails and is financially reverted. Second, regardless of the stage at which the execution fails, $\mathcal{F}_{\mathsf{UIP}}$ holds the misbehaved parties accountable for the failure. Third, if $\mathcal{F}_{\mathsf{blockchain}}$ is modeled with bounded transaction finality latency, Op is guaranteed to finish correctly if both parties are honest. Finally, $\mathcal{F}_{\mathsf{UIP}}$, by design, makes the *contract* public. This is because in the real world protocol $\mathsf{Prot}_{\mathsf{UIP}}$, the status of execution is public both on the ISC and the NSB. We acknowledge such limitation in this paper and leave the support for privacy-preserving blockchain interoperability to future work.

```
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 CloseureClaim($\widetilde{T}$, [ChainID, StateRoot, TxRoot ]):
  (_, _, StatusPool) := Data[Epoch]; abort if not found
  update StatusPool.Add($\widetilde{T}$)
  # Protocol of an individual honest peer $\mathcal{V}$ to ensure correctness
  Init: $\mathcal{V}$.StatusPool := []
  Daemon Watching($\widetilde{T}$, $\mathcal{S}$ = {$\text{Sig}^{\text{peer}}$, ...})
    abort if $\widetilde{T}$ is already in $\mathcal{V}$.StatusPool
    abort if $\mathcal{S}$ contains more than $\mathcal{K}$ distinguished signatures
    abort if $\widetilde{T}$ is not finalized on its destination blockchain (ChainID)
    abort if the reported StateRoot and TxRoot are not authentic
    update $\mathcal{V}$.StatusPool.Add($\widetilde{T}$)
    update $\mathcal{S}$.Add(Cert([$\widetilde{T}$, [ChainID, StateRoot, TxRoot]]; $\text{Sig}^{\mathcal{V}}$))
    multicast ($\widetilde{T}$, $\mathcal{S}$) to other peers of the NSB
Daemon CloseureWatching({$\text{Prot}_{\text{BC}}$, ...}):        Proactive Streaming Version
  proactively watch $\text{Prot}_{\text{BC}}$ for recently finalized blocks {$\mathcal{B}$, ...}
  (_, _, StatusPool) := Data[Epoch]; abort if not found
  retrieve the root $R_{tx}$ of TxMT and $R_{state}$ of StateMT on $\mathcal{B}$
  update StatusPool.Add($R_{tx}$, $R_{state}$)
  # Protocol of an individual honest peer $\mathcal{V}$ to ensure correctness
  Init: $\mathcal{V}$.StatusPool := []
  Daemon Watching({$\text{Prot}_{\text{BC}}$, ...}) and Watching($\mathcal{B}$, $\mathcal{S}$ = {Sig, ...})
    abort if $\mathcal{B}$ is not finalized on $\text{Prot}_{\text{BC}}$ 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 $R_{state}$ of StateMT on $\mathcal{B}$
    $\mathcal{S}$.Add(Cert([$R_{tx}$, $R_{state}$]; $\text{Sig}^{\mathcal{V}}$)); $\mathcal{V}$.StatusPool.Add($R_{tx}$, $R_{state}$)
    multicast ($\mathcal{B}$, $\mathcal{S}$) to other peers of the NSB
Upon Receive AddAction(Cert):
  (_, ActionPool, _) := Data[Epoch]; abort if not found
  update ActionPool.Add(Cert)
  # 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 10: Detailed protocol description of Prot_{NSB}. Interfaces annotated with override are also implemented by Prot_{BC}. Gray background denotes the protocol of honest peers in the NSB to ensure the correctness for the corresponding interface.

A.2.3 Main Security Theorems

In this section, we claim the main security theorem of HyperService. The correctness of Theorem A.1 guarantees that Prot_{UIP} achieves same security properties as $\mathcal{F}_{\text{UIP}}$.

Theorem A.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 Prot_{UIP} securely UC-realizes the ideal functionality $\mathcal{F}_{\text{UIP}}$ against a malicious adversary in the passive corruption model.*

We further consider a variant of Prot_{UIP}, referred to as H-Prot_{UIP}, that requires $\mathcal{P}_{\text{VES}}$ and $\mathcal{P}_{\text{CLI}}$ to *only use* $\mathcal{P}_{\text{NSB}}$ as the communication medium.

Theorem A.2. *With the same assumption of Theorem A.1, the UIP protocol variant H-Prot_{UIP} securely UC-realizes the ideal functionality $\mathcal{F}_{\text{UIP}}$ against a malicious adversary in the Byzantine corruption model.*

A.2.4 Proof Overview

We now the prove our main theorems. We start with Theorem A.1. In the UC framework [25], 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 *passive* corruption model (used by Theorem A.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.2.7 discusses the *Byzantine* corruption model, where the adversary takes complete control of the corrupted party.

To prove that Prot_{UIP} UC-realizes the ideal functionality $\mathcal{F}_{\text{UIP}}$, we need to prove that Prot_{UIP} *UC-emulates* $\mathcal{I}_{\mathcal{F}_{\text{UIP}}}$, which is the *ideal protocol* (defined below) of our ideal functionality $\mathcal{F}_{\text{UIP}}$. 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 $(\mathcal{I}_{\mathcal{F}_{\text{UIP}}}, \mathcal{S})$, and the real world, featured by $(\text{Prot}_{\text{UIP}}, \mathcal{A})$. Mathematically, on any input, the probability that $\mathcal{E}$ outputs $\overrightarrow{1}$ after interacting with $(\text{Prot}_{\text{UIP}}, \mathcal{A})$ in the real world differs by at most a negligible amount from the probability that $\mathcal{E}$ outputs $\overrightarrow{1}$ after interacting with $(\mathcal{I}_{\mathcal{F}_{\text{UIP}}}, \mathcal{S})$ in the ideal world.

The ideal protocol $\mathcal{I}_{\mathcal{F}_{\text{UIP}}}$ is a wrapper around $\mathcal{F}_{\text{UIP}}$ by a set of dummy parties that have the same interfaces as the parties of Prot_{UIP} in the real world. As a result, $\mathcal{E}$ is able to interact with $\mathcal{I}_{\mathcal{F}_{\text{UIP}}}$ in the ideal world the same way it interacts with Prot_{UIP} in the real world. These dummy parties simply pass received input from $\mathcal{E}$ to $\mathcal{F}_{\text{UIP}}$ and relay output of $\mathcal{F}_{\text{UIP}}$ to $\mathcal{E}$, without implementing

```
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]; $\mathsf{Sig}^{\mathcal{P}_z}_{\mathsf{sid}}$, $\mathsf{Sig}^{\mathcal{P}_a}_{\mathsf{sid}}$) 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 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 $\mathcal{T}$.state := init
    compute $\mathsf{Cert}^i_{\mathcal{T}}$ := Cert([$\mathcal{T}$, init, sid]; $\mathsf{Sig}^{\mathcal{P}_z}_{\mathsf{sid}}$)
    send $\mathsf{Cert}^i_{\mathcal{T}}$ to both {$\mathcal{P}_a$, $\mathcal{P}_z$} to inform action
Upon Receive ReqTransInited($\mathcal{T}$, sid, $\mathcal{P}$)
    ($\mathcal{G}_T$, _, _, _) := Data[sid]; abort if not found
    assert $\mathcal{P}$ = $\mathcal{T}$.from and $\mathcal{T}$.state = init
    compute the on-chain transaction $\widetilde{T}$ for $\mathcal{T}$
    update $\mathcal{T}$.state := inited and $\mathcal{T}$.trans := $\widetilde{T}$
    compute $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ := Cert([$\widetilde{T}$, inited, sid, $\mathcal{T}$]; $\mathsf{Sig}^{\mathcal{P}}_{\mathsf{sid}}$)
    send $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ 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, $\mathcal{T}$.state = inited and $\mathcal{T}$.trans = $\widetilde{T}$
    update $\mathcal{T}$.state := open and get $\mathsf{ts}_{\mathsf{open}}$ := now()
    compute $\mathsf{Cert}^o_{\mathcal{T}}$ := Cert([$\widetilde{T}$, open, $\mathsf{ts}_{\mathsf{open}}$, sid, $\mathcal{T}$]; $\mathsf{Sig}^{\mathcal{P}}_{\mathsf{sid}}$)
    send $\mathsf{Cert}^o_{\mathcal{T}}$ 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, $\mathcal{T}$.state = open and $\mathcal{T}$.trans = $\widetilde{T}$
    assert $\mathsf{ts}_{\mathsf{open}}$ is within the error boundary with now()
    update $\mathcal{T}$.state = 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 $\mathsf{Cert}^{\mathsf{od}}_{\mathcal{T}}$ := 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 $\mathsf{Cert}^{\mathsf{od}}_{\mathcal{T}}$ 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 $\mathcal{T}$.state = opened and $\mathcal{T}$.trans = $\widetilde{T}$
    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 current time now()
    update $\mathcal{T}$.state := closed and $\mathcal{T}.\mathsf{ts}_{\mathsf{closed}}$ := $\mathsf{ts}_{\mathsf{closed}}$
    compute $\mathsf{Cert}^c_{\mathcal{T}}$ := 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 $\mathsf{Cert}^c_{\mathcal{T}}$ to both {$\mathcal{P}_a$, $\mathcal{P}_z$} to inform action
Upon Receive TermExecution(sid, $\mathcal{P} \in (\mathcal{P}_a, \mathcal{P}_z)$) 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 $\mathcal{T}$.state = unknown : update resp[$\mathcal{T}$] := $\mathcal{P}_z$
        elif $\mathcal{T}$.state = init : update resp[$\mathcal{T}$] := $\mathcal{P}_a$
        elif $\mathcal{T}$.state = inited : update resp[$\mathcal{T}$] := $\mathcal{T}$.to
        elif $\mathcal{T}$.state = open and $\mathcal{T}$.state = opened :
            update resp[$\mathcal{T}$] := $\mathcal{T}$.from
        elif $\mathcal{T}$.state = closed and deadline constraint fails :
            update resp[$\mathcal{T}$] =: $\mathcal{T}$.from
    financially revert all closed transactions if resp is not empty
    return any remaining funds in stake to corresponding senders
    remove the internal bookkeeping of sid from Data
```

Figure 11: The ideal functionality $\mathcal{F}_{\mathsf{UIP}}$.

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 $\{\mathcal{P}_{\mathsf{VES}}, \mathcal{P}_{\mathsf{CLI}}, \mathcal{P}_{\mathsf{ISC}}, \mathcal{P}_{\mathsf{NSB}}, \mathcal{P}_{\mathsf{BC}}\}$. 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 [25], 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.2.5 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 $\mathcal{P}^{\mathcal{I}}_{\mathsf{CLI}}$ and $\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$, $\mathcal{S}$ emulates the $\mathcal{G}_T$ and *contract* setup (*c.f.*, § A.2.6) and constructs a SessionCreate call to $\mathcal{F}_{\mathsf{UIP}}$ with parameter ($\mathcal{G}_T$, *contract*, $\mathcal{P}^{\mathcal{I}}_{\mathsf{CLI}}$, $\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}$).
- 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 ensure the association between $\mathcal{T}$ and $\widetilde{T}$.

- Due to the asymmetry of interfaces defined by $\mathcal{P}^{\mathcal{I}}_{\mathsf{CLI}}$ 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.2.6 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 [25], 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 $(\mathcal{P}_{\mathsf{VES}}, \mathcal{P}_{\mathsf{CLI}})$ pair is replaced by the dummy $(\mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}, \mathcal{P}^{\mathcal{I}}_{\mathsf{CLI}})$ pair. Upon observing an instruction from $\mathcal{E}$ to execute some dApp executables $\mathcal{G}_T$, $\mathcal{S}$ calls the CreateContract interface of $\mathcal{P}_{\mathsf{ISC}}$ (living in the Hybrid A_1) to obtain the contract code *contract*. Upon *contract* is received, $\mathcal{S}$ calls the SessionCreate interface of $\mathcal{F}_{\mathsf{UIP}}$ with parameter $(\mathcal{G}_T, \mathit{contract}, \mathcal{P}^{\mathcal{I}}_{\mathsf{VES}}, \mathcal{P}^{\mathcal{I}}_{\mathsf{CLI}})$, which will output a certificate to both dummy parties to emulate the handshake result between $\mathcal{P}_{\mathsf{VES}}$ and $\mathcal{P}_{\mathsf{CLI}}$ 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.2.5, $\mathcal{S}$ has enough information to construct a call to $\mathcal{F}_{\mathsf{UIP}}$ with a proper interface and parameters. If the call generates a certificate Cert, $\mathcal{S}$ retrieves Cert to emulate the PoAs staking in the real world. In particular, if in the real world, $\mathcal{P}_{\mathsf{VES}}$ (and $\mathcal{P}_{\mathsf{CLI}}$) publishes a certificate on $\mathcal{P}_{\mathsf{NSB}}$ after receiving the same instruction from $\mathcal{E}$, then $\mathcal{S}$ publishes the corresponding certificate on $\mathcal{P}_{\mathsf{NSB}}$ in the Hybrid A_1 as well. Otherwise, $\mathcal{S}$ skip the publishing. Later, $\mathcal{S}$ retrieves (and stores) the Merkle proof from $\mathcal{P}_{\mathsf{NSB}}$, and then instructs $\mathcal{F}_{\mathsf{UIP}}$ to output the proof to the dummy party which, from the point view of $\mathcal{E}$, should be the publisher of Cert.

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

Note that in the real world, the execution of $\mathcal{G}_T$ is automatic in the sense that $\mathcal{G}_T$ can continuously proceed even without additional instructions from $\mathcal{E}$ after successful session setup. In the Hybrid A_1, although $\mathcal{P}_{\mathsf{VES}}$ and $\mathcal{P}_{\mathsf{CLI}}$ are replaced by dummy parties, $\mathcal{S}$, with fully knowledge of $\mathcal{G}_T$, is still able to drive the execution of $\mathcal{G}_T$ so that from $\mathcal{E}$'s perspective, $\mathcal{G}_T$ 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 execute a $\mathcal{G}_T$, $\mathcal{S}$ computes the arbitration code *contract*, 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 $(\mathsf{sid}, \mathcal{P}^{\mathcal{I}}_{\mathsf{CLI}})$ 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.2.7 Byzantine Corruption Model

Theorem A.1 considers the passive corruption model. In this section, we discuss the more general Byzantine corruption model for $\mathcal{P}_{\mathsf{VES}}$ and $\mathcal{P}_{\mathsf{CLI}}$ (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, a Byzantine-corrupted party may send conflicting messages to off-chain channels and $\mathcal{P}_{\mathsf{NSB}}$. Note that for any transaction state, $\mathsf{Prot}_{\mathsf{UIP}}$ always processes the first received attestation (either a certificate from channels or Merkle proof from the $\mathcal{P}_{\mathsf{NSB}}$) and effectively ignores the other one. The adversary could then inject message inconsistency to make the protocol execution favors one type of attestations over the other. This makes it impossible for $\mathcal{S}$ to always accurately emulate its behaviors, 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 $\mathcal{P}_{\mathsf{CLI}}$ to *only use* $\mathcal{P}_{\mathsf{NSB}}$ as the communication medium. Thus, the full granularity of protocol execution is guaranteed to be unique, allowing $\mathcal{S}$ to emulate whatever actions a (corrupted) part may take in the real world. Therefore, it is not hard to conclude the Theorem A.2.

A.3 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 [15, 28]), 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 [27] capability for privacy-preserving computation [29, 54, 61]. 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 [39, 40, 47]. 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, HyperService 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.