HyperService: Interoperability and Programmability Across Heterogeneous Blockchains

Zhuotao Liu[1,2] Yangxi Xiang[3] Jian Shi[4] Peng Gao[5] Haoyu Wang[3]
Xusheng Xiao[4,2] Bihan Wen[6] Yih-Chun Hu[1,2]
[1]University of Illinois at Urbana-Champaign [2]HyperService Consortium
[3]Beijing University of Posts and Telecommunications [4]Case Western Reserve University
[5]University of California, Berkeley [6]Nanyang Technological University
hyperservice.team@gmail.com

ABSTRACT

Blockchain interoperability, which allows state transitions across different blockchain networks, is critical functionality to facilitate major blockchain adoption. Existing interoperability protocols mostly focus on atomic token exchanges between blockchains. However, as blockchains have been upgraded from passive distributed ledgers into programmable state machines (thanks to smart contracts), the scope of blockchain interoperability goes beyond just token exchanges. 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 approximately 35,000 lines of code to demonstrate its practicality. Our experiments show that (i) HyperService imposes reasonable latency, in order of seconds, on the end-to-end execution of cross-chain applications; (ii) the HyperService platform is scalable to continuously incorporate new large-scale production blockchains.

CCS CONCEPTS

• **Security and privacy → Distributed systems security**; **Security protocols**.

KEYWORDS

Blockchain Interoperability; Smart Contract; Cross-chain dApps

ACM Reference Format:
Zhuotao Liu[1,2] Yangxi Xiang[3] Jian Shi[4] Peng Gao[5] Haoyu Wang[3] and Xusheng Xiao[4,2] Bihan Wen[6] Yih-Chun Hu[1,2]. 2019. HyperService: Interoperability and Programmability Across Heterogeneous Blockchains. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19), November 11–15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3319535.3355503

CCS '19, November 11–15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6747-9/19/11…$15.00
https://doi.org/10.1145/3319535.3355503

1 INTRODUCTION

Over the last few years, we have witnessed rapid growth of several flagship blockchain applications, such as the payment system Bitcoin [53] and the smart contract platform Ethereum [27]. Since then, considerable effort has been made to improve the performance and security of individual blockchains, such as more efficient consensus algorithms [3, 8, 32, 43], improving transaction rate by sharding [20, 44, 51, 60] and payment channels [37, 40, 52], enhancing the privacy for smart contracts [29, 39, 45], and reducing their vulnerabilities via program analysis [24, 46, 50].

As a result, in today's blockchain ecosystem, we see many distinct blockchains, falling roughly into the categories of public, private, and consortium blockchains [6]. In a world deluged with isolated blockchains, interoperability is power. Blockchain interoperability enables secure state transitions across different blockchains, which is invaluable for connecting the decentralized Web 3.0 [26]. Existing interoperability proposals [21, 36, 38, 61] mostly center around atomic token exchange between two blockchains, aiming to eliminate the requirement of centralized exchanges. However, since smart contracts executing on blockchains have transformed blockchains from append-only distributed ledgers into programmable state machines, we argue that *token exchange is not the complete scope of blockchain interoperability.* Instead, blockchain interoperability is complete only with *programmability*, allowing developers to write decentralized applications executable across those disconnected state machines.

We recognize at least 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 [59]. This, however, raises a virtualization challenge to abstract away the heterogeneity of smart contracts and accounts on different blockchains so that the interactions and operations among those contracts and accounts can be *uniformly* specified when writing cross-chain dApps.

Second, existing token-exchange oriented interoperability protocols, such as atomic cross-chain swaps (ACCS) [5], are not generic enough to realize cross-chain dApps. This is because the "executables" of those dApps could contain more complex operations than token transfers. For instance, our example dApp in § 2.3 invokes a smart contract using parameters obtained from smart contracts

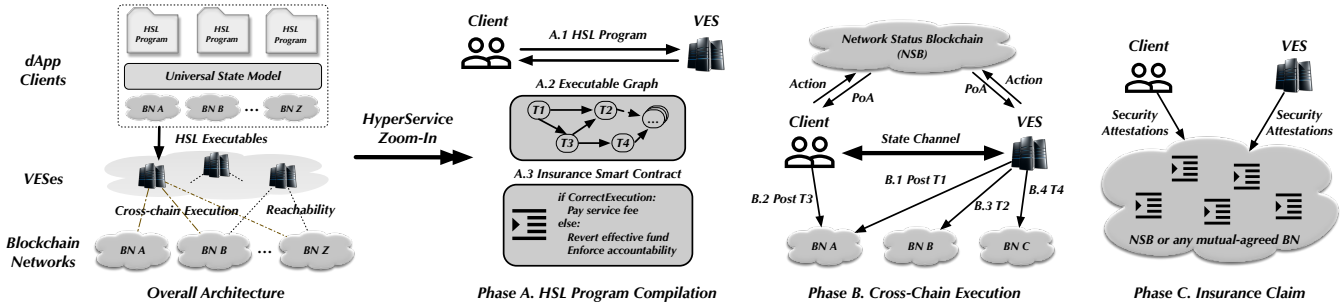

Figure 1: The architecture of HyperService.

deployed on different blockchains. The complexity of this operation is far beyond mere token transfers. In addition, the executables of cross-chain dApps often contain transactions on different blockchains, and the correctness of dApps requires those transactions to be executed with certain preconditions and deadline constraints. Another technical challenge is to securely coordinate those transactions to enforce dApp correctness in a fully decentralized manner with zero trust assumptions.

To meet these challenges, we propose HyperService, the first platform for building and executing dApps across heterogeneous blockchains. At a very high level, HyperService is powered by two innovative designs: a developer-facing *programming framework* for writing cross-chain dApps, and a blockchain-facing cryptography protocol to securely realize those dApps on blockchains. Within this 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 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 parties are held accountable, and (iii) *financially atomic*, meaning all involved parties experience almost zero financial losses, regardless of the execution status of dApps. UIP is fully trust-free, 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. Specifically, we make the following major contributions in this paper.

(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 high-level language to describe and program dApps. Using our framework, a developer 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 cross-chain token exchanges. Rather, UIP is capable of securely realizing complex cross-chain operations that involve smart contracts deployed on heterogeneous blockchains. We express the security properties of UIP via an ideal functionality $\mathcal{F}_{\mathsf{UIP}}$ and rigorously prove that UIP realizes $\mathcal{F}_{\mathsf{UIP}}$ in the Universal Composability (UC) framework [28].

(iii) We implement a prototype of HyperService in approximately 35,000 lines of code, and evaluate the prototype with three categories of cross-chain dApps. Our experiments show that the end-to-end dApp execution latency imposed by HyperService is in the order of seconds, and the HyperService platform has sufficient capacity to continuously incorporate new production blockchains.

2 HYPERSERVICE OVERVIEW

2.1 Architecture

As depicted in Figure 1, architecturally, HyperService consists of 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 dApp clients employ the underlying UIP cryptography protocol to securely execute those transactions 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* designed by HyperService to provide an objective and unified view of the dApps' execution status, based on which the ISCs arbitrate the correctness or violation of dApp executions in a trust-free manner. 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, together with smart contracts (or dApps) executed on the blockchain, is often perceived as a state machine [59]. We desire to preserve the similar abstraction for developers when writing cross-chain dApps. Towards this end, we propose Unified State Model (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

Table 1: Example of entities, operations and dependencies in USM

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

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 $\mathcal{M} = \{\mathcal{E}, \mathcal{P}, C\}$ where $\mathcal{E}$ is a set of *entities*, $\mathcal{P}$ is a set of *operations* performed over those entities, and C is a set of constraints defining the *dependencies* 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 (we discuss the extensions of USM in § 6.1). 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 public attributes, such as state variables, callable interfaces, and its source code deployed on blockchains. Entity attributes are crucial to enforce the security and correctness of dApps during compilation, as discussed in § 2.3.

An operation in USM defines a step of computation performed over several entities. Table 1 lists two 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 a method specified by the interface of a contract entity using compatible parameters, whose values may 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 consensus processes are not synchronized. To honor the possible dependencies among events in distributed computing [47], USM, therefore, defines constraints to specify dependencies among operations. Currently, USM supports two kinds of dependencies: *preconditions* and *deadlines*, where an operation can proceed only if all its preconditioning operations are finished, and an operation must be finished within a bounded time interval 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, where the state of upstream nodes is persistent and can be used by downstream nodes; (ii) deadlines are crucial to ensure the forward progress of dApp executions.

2.3 HyperService Programming Language

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

2.3.1 An Introductory Example for HSL Programs

Financial derivatives are among the most commonly cited blockchain applications. Many financial derivatives rely on authentic data feed, *i.e.,* an *oracle*, as inputs. For instance, a standard call-option contract needs a genuine strike price. Existing oracles [13, 62] 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 smart contract on each individual blockchain since synchronizing the execution of those oracle 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 cryptocurrencies.

```
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.

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 cryptocurrencies they prefer. The code snippet shown in Figure 2 is the HSL implementation for the referred dApp. In this dApp, both Option contracts deployed on blockchains *ChainY* and *ChainZ* rely on the same Broker contract on *ChainX* to provide the genuine strike price (lines 14 and 15 in 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 checks 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 smart 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 checks 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 languages different from that of the invoking contract.

Once a HSL program passes the syntax and correctness checks, the compiler will generate an *executable* for the program. The executable is structured in the form of a Transaction Dependency Graph, which contains (i) the complete information for computing a set of blockchain-executable transactions, (ii) the metadata of each transaction needed 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 its reachability (*i.e.,* the set of blockchains that the VES supports), service fees charged for correct executions, and insurance plans (*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 § 6.3, we lay out our visions for VESes.

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

2.4 Universal Inter-Blockchain Protocol (UIP)

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 executions on different 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 contract execution environment. UIP provides strong security guarantees 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 finalized 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 transaction-status proofs constructed from the NSB as input to determine the correctness or violation of dApp executions, and meanwhile uses action proofs to determine the accountable parities in case of exceptions.

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 [28].

2.5 Assumptions and Threat Model

We assume that the cryptographic primitives and the consensus protocol of all underlying blockchains are secure so that each of them can have the concept of *transaction finality*. On Nakamoto consensus based blockchains (typically permissionless), this is achieved by assuming that the probability of blockchain reorganizations drops exponentially as new blocks are appended (*common-prefix property*) [35]. On Byzantine tolerance based blockchains (usually permissioned), finality is guaranteed by signatures from a quorum of permissioned voting nodes. For a blockchain, if the NSB-proposed definition of transaction finality for the blockchain is accepted by users and dApps on HyperService, the operation (or trust) model (*e.g.,* permissionless or permissioned) and consensus efficiency (*i.e.,* the latency for a transaction to become final) of the blockchain have provably no impact on the security guarantees of our UIP protocol. We also assume that each underlying blockchain has a public ledger that allows external parties to examine and prove transaction finality and the *public* state of smart contracts.

The correctness of UIP relies on the correctness of the NSB. An example implementation of NSB is a permissioned blockchain, where any information on NSB becomes legitimate only if a quorum of consensus nodes that maintain the NSB have approved the information. We thus assume that at least $\mathcal{K}$ consensus nodes of the NSB are honest, where $\mathcal{K}$ is the quorum threshold (*e.g.,* the majority). In this design, an NSB node is not required to become either a full or light node for any of the underlying blockchains.

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

3 PROGRAMMING FRAMEWORK

The design of the 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 interoperable entities defined in the HSL program. Afterwards, the compiler performs semantic validations on

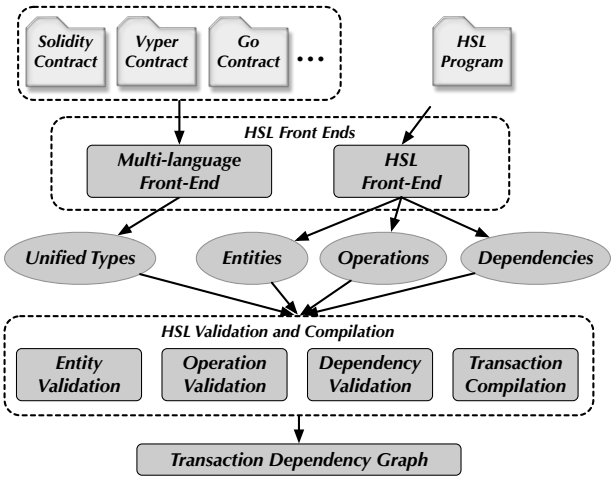

Figure 3: Workflow of HSL compilation.

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 the form of a transaction dependency graph. We next describe the details of each component.

3.1 Unified Type System

The USM is designed 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 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, as shown in Table 2. Data types that are commonly used in smart contract programming languages will be *mapped* to 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 via 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 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 types in our unified type system. 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. Our unified type system is horizontally scalable to support additional strong-typed programming languages. Note that the use of complex data types as contract function parameters has not been fully supported yet in production. We thus leave complex types in HSL to future work.

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

⟨*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

3.2 HSL Language Design

The language constructs provided by HSL are coherent with 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*⟩). Note that no new coins on any blockchains are ever created during the operation. 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).

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 constraints of each operation. The deadline dependency may be given either using the number of blocks on NSB (⟨*int*⟩ *blocks*) or in absolute time (⟨*int*⟩ ⟨*time_unit*⟩), 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 acyclic 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 $\mathcal{G}_T$. Each vertex of $\mathcal{G}_T$, referred to as a *transaction wrapper*, contains the complete information to compute an on-chain transaction executable on a specific blockchain, as well as additional metadata for 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 show 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

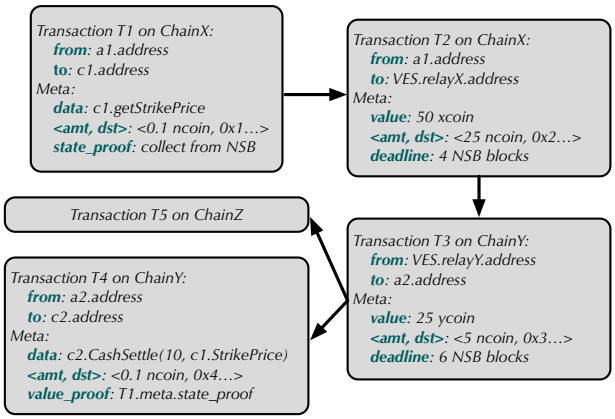

Figure 4: $\mathcal{G}_T$ generated for the example HSL program.

the fields of meta. First, to achieve financial atomicity, meta must populate a tuple ⟨amt, dst⟩ 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 *op*1 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 in general VES-specific.

Finally, the deadlines of transactions could be specified using the number of blocks on the NSB. This is because the NSB constructs a unified view of the status of all underlying blockchains and therefore can measure the execution time of each transaction. Specifically, the deadline of a transaction $\mathcal{T}$ is measured as the number of blocks between two NSB blocks $\mathcal{B}_1$ and $\mathcal{B}_2$ (including $\mathcal{B}_2$), where $\mathcal{B}_1$ proves the finalization of $\mathcal{T}$'s last preconditioned transaction and $\mathcal{B}_2$ proves the finalization of $\mathcal{T}$ itself. We explain in detail how the finality proof is constructed based on NSB blocks in § 4.2.2. Transaction deadlines are indeed enforced by the ISC using the number of NSB blocks. Note that to improve expressiveness, the HSL language also allows developers to define deadlines in time intervals (*e.g.,* minutes). The compiler will then convert those time intervals into numbers of NSB blocks.

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 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 (except for 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 *fallback* 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 in case of exceptions, eliminating the ambiguity for accountability enforcement.

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 Prot_{CLI} 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. Similar to typical blockchain blocks, an NSB block contains several common 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.

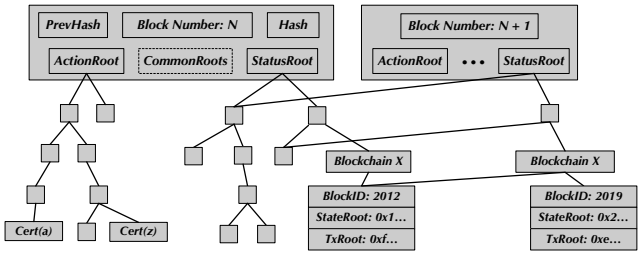

Figure 5: The architecture of NSB blocks.

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 based on the TxRoots and StateRoots retrieved from the blockchain's public ledger. Although the exact namings may vary on different blockchains, in general, the TxRoot and StateRoot in a blockchain block represent the root of a Merkle tree storing transactions and storage state (*e.g.,* account balance, contract state), respectively. Note that the NSB only stores *relevant* blockchain state, where a blockchain block is considered to be relevant if the block 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 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 of each ActionMT is static, we lexicographically sort the ActionMT to achieve fast search and convenient proof of non-membership.

Note that the construction of StatusMT ensures that each underlying blockchain can have a dedicated subtree for storing its transaction status. This makes the NSB *shardable on the granularity of individual blockchains*, ensuing that the NSB is horizontally scalable as HyperService continuously incorporates additional blockchains. Prot_{NSB}, discussed in § 4.5, 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 Prot_{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, Prot_{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, Prot_{VES} interacts with the protocol Prot_{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, Cert([∗]; Sig) represents a signed certificate proving that the signing party agrees on the value enclosed in the certificate. We use $\text{Sig}^{\mathcal{V}}_{\text{sid}}$ and $\text{Sig}^{\mathcal{D}}_{\text{sid}}$ to represent the signature by Prot_{VES} and Prot_{CLI}, respectively.

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

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

Additionally, both Prot_{VES} and Prot_{CLI} are required to deposit sufficient funds to Prot_{ISC} to ensure that Prot_{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 $\mathcal{X}$ (Prot_{VES} or Prot_{CLI}) only needs to stake

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

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

After the *contract* is instantiated and sufficiently staked, Prot_{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 Prot_{NSB} and Prot_{BC}.

4.2.2 Protocol Exchange for Transaction Handling

In Prot_{VES}, SInitedTrans and OpenTrans are two handlers processing *northbound* transactions which originates from Prot_{VES}. The SInitedTrans handling for $\mathcal{T}$ is invoked when all its preconditions are finalized, which is detected by the watching service of Prot_{VES} (*c.f.*, § 4.2.3). The SInitedTrans computes $\text{Cert}^{\text{id}}_{\mathcal{T}}$ to prove $\mathcal{T}$ is in the inited state , and then passes it to the corresponding handler of Prot_{CLI} for subsequent processing. Meanwhile, SInitedTrans stakes $\text{Cert}^{\text{id}}_{\mathcal{T}}$ on Prot_{NSB}, and later it retrieves a Merkle proof $\text{Merk}^{i}_{\mathcal{T}}$ from the NSB to prove that $\text{Cert}^{\text{id}}_{\mathcal{T}}$ has been sent. $\text{Merk}^{\text{id}}_{\mathcal{T}}$ essentially is a hash chain linking $\text{Cert}^{\text{id}}_{\mathcal{T}}$ 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 $\text{Cert}^{o}_{\mathcal{T}}$, which is supposed to be generated by Prot_{CLI} after it processes $\text{Cert}^{\text{id}}_{\mathcal{T}}$ from Prot_{VES}. OpenTrans performs special correctness check on the ts_{open} enclosed in $\text{Cert}^{o}_{\mathcal{T}}$. In particular, Prot_{VES} and Prot_{CLI} use the block height of the NSB as a calibrated clock. By checking that ts_{open} is within a bounded range of the NSB height, Prot_{VES} ensures that the ts_{open} added by Prot_{CLI} is fresh. After all correctness checks on $\text{Cert}^{\text{id}}_{\mathcal{T}}$ 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 the $\text{Cert}^{o}_{\mathcal{T}}$ received from Prot_{CLI} and a post-open (*i.e.,* opened) certificate $\text{Cert}^{\text{od}}_{\mathcal{T}}$ computed by Prot_{VES} is that latter one is signed by both parties. Only the ts_{open} specified in $\text{Cert}^{\text{od}}_{\mathcal{T}}$ is used by Prot_{ISC} when evaluating the deadline constraint of $\mathcal{T}$.

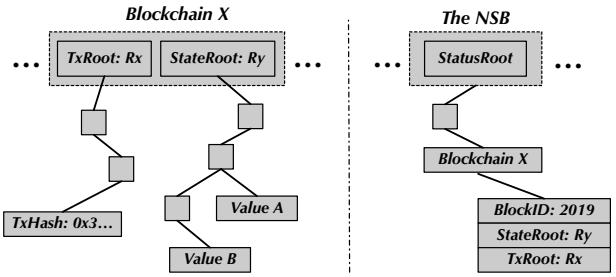

Figure 7: The complete on-chain proof (denoted by $\text{Merk}^{c}_{\mathcal{T}}$) to prove that the state of a transaction is eligible to be promoted as closed. The left-side part is the finalization proof (denoted by $\text{Merk}^{c_1}_{\mathcal{T}}$) for the transaction collected from its destination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

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

Cross-chain execution makes forward progress when all session-relevant blockchains and the NSB make progress on transactions. 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

Init: Data := ∅
Upon Receive CreateContract($\mathcal{G}_T$):
  generate the arbitration cod, denoted by *contract*, as follows
  initialize three maps $\mathsf{T}_{\mathsf{state}}$, $\mathsf{A}_{\mathsf{revs}}$ and $\mathsf{F}_{\mathsf{stake}}$
  for each $\mathcal{T} \in \mathcal{G}_T$ **:**
    compute an internal identifier for $\mathcal{T}$ as tid := $H(\mathcal{T})$
    initialize $\mathsf{T}_{\mathsf{state}}$[tid] := [unknown, $\mathcal{T}$, $\mathsf{ts}_{\mathsf{open}}$=0, $\mathsf{ts}_{\mathsf{closed}}$=0, $\mathsf{st}_{\mathsf{proof}}$]
    retrieve tid's fund-reversion account, denoted as dst
    initialize $\mathsf{A}_{\mathsf{revs}}$[tid] := [amt=0, dst]
  compute an identifier for *contract* as cid := $H(\overrightarrow{0}$, *contract*)
  initialize Data[cid] := [$\mathcal{G}_T$, $\mathsf{T}_{\mathsf{state}}$, $\mathsf{A}_{\mathsf{revs}}$, $\mathsf{F}_{\mathsf{stake}}$]
  send [cid, *contract*] to the requester for acknowledgment
Upon Receive StakeFund(cid):
  (_, _, _, _, $\mathsf{F}_{\mathsf{stake}}$) := Data[cid]; abort if not found
  update $\mathsf{F}_{\mathsf{stake}}$[msg.sender] := $\mathsf{F}_{\mathsf{stake}}$[msg.sender] + msg.value
Upon Receive InsuranceClaim(cid, Atte):
  (_, _, $\mathsf{T}_{\mathsf{state}}$, _, _) := Data[cid]; abort if not found
  compute tid := H(Atte.$\mathcal{T}$); T := $\mathsf{T}_{\mathsf{state}}$[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 $\mathsf{Cert}^{\mathsf{od}}_{\mathcal{T}}$ **:** update T.state := opened; T.$\mathsf{ts}_{\mathsf{open}}$:= Atte.$\mathsf{ts}_{\mathsf{open}}$
    else : update T.state := closed; T.$\mathsf{ts}_{\mathsf{closed}}$:= Atte.$\mathsf{ts}_{\mathsf{closed}}$
  else : # Atte is in form of a Merkle proof
    assert MerkleVerify(Atte) is true
    if Atte is a $\mathsf{Merk}^{i}_{\mathcal{T}}$ **or** $\mathsf{Merk}^{\mathsf{id}}_{\mathcal{T}}$ **or** $\mathsf{Merk}^{o}_{\mathcal{T}}$ **:**
      retrieve the certificate $\mathsf{Cert}^{i}_{\mathcal{T}}$ or $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ or $\mathsf{Cert}^{o}_{\mathcal{T}}$ from Atte
      assert the $\widetilde{T}$ enclosed in $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ or $\mathsf{Cert}^{o}_{\mathcal{T}}$ is genuine
      assert the $\mathsf{ts}_{\mathsf{open}}$ enclosed in $\mathsf{Cert}^{o}_{\mathcal{T}}$ is genuine
      update T.state := Atte.state
    elif Atte is $\mathsf{Merk}^{\mathsf{od}}_{\mathcal{T}}$ **:**
      retrieve the certificate $\mathsf{Cert}^{\mathsf{od}}_{\mathcal{T}}$ from Atte
      update T.state := opened and T.$\mathsf{ts}_{\mathsf{open}}$:= $\mathsf{Cert}^{\mathsf{od}}_{\mathcal{T}}$.$\mathsf{ts}_{\mathsf{open}}$
    elif Atte is $\mathsf{Merk}^{c}_{\mathcal{T}}$ **:**
      update T.$\mathsf{st}_{\mathsf{proof}}$ based on $\mathsf{Merk}^{c_1}_{\mathcal{T}}$ if necessary
      update T.$\mathsf{ts}_{\mathsf{closed}}$ as the height of the block attaching $\mathsf{Merk}^{c_2}_{\mathcal{T}}$
      update T.state := closed
Upon Timeout SettleContract(cid): *Internal Daemon*
  ($\mathcal{G}_T$, $\mathsf{T}_{\mathsf{state}}$, $\mathsf{A}_{\mathsf{revs}}$, $\mathsf{F}_{\mathsf{stake}}$) := Data[cid]; abort if not found
  for (tid, T) ∈ $\mathsf{T}_{\mathsf{state}}$ **:**
    continue if T.state is not closed
    update $\mathsf{A}_{\mathsf{revs}}$[tid].amt := T.$\mathcal{T}$.meta.amt
    if DeadlineVerify(T) = **true :** update T.state := correct
  compute $\mathcal{S}$:= DirtyTrans($\mathcal{G}_T$, $\mathsf{T}_{\mathsf{state}}$) # non-empty if execution fails.
  execute fund reversion for non-zero entries in $\mathsf{A}_{\mathsf{revs}}$ if $\mathcal{S}$ is not empty
  initialize a map *resp* to record which party to blame
  for each (tid, T) ∈ $\mathcal{S}$ **:**
    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 $\mathsf{F}_{\mathsf{stake}}$ to corresponding senders
  call Data.erase[cid] to stay silent afterwards

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

$\mathsf{Prot}_{\mathsf{CLI}}$), $\mathsf{Prot}_{\mathsf{VES}}$ tries to retrieve the second part of $\mathsf{Merk}^{c}_{\mathcal{T}}$ from $\mathsf{Prot}_{\mathsf{NSB}}$. The second proof, denoted as $\mathsf{Merk}^{c_2}_{\mathcal{T}}$, is to prove that the Merkle roots referred in $\mathsf{Merk}^{c_1}_{\mathcal{T}}$ are correctly linked to a StatusRoot on a finalized NSB block (see Figure 7). Once $\mathsf{Merk}^{c}_{\mathcal{T}}$ is fully constructed, the state of $\mathcal{T}$ is promoted as closed. Finally, $\mathsf{Prot}_{\mathsf{VES}}$ 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, $\mathsf{Prot}_{\mathsf{VES}}$ processes them by either requesting initialization from $\mathsf{Prot}_{\mathsf{CLI}}$ or calling SInitedTrans internally, depending on the originators of those transactions.

4.2.4 $\mathsf{Prot}_{\mathsf{ISC}}$ Invocation

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

4.3 Execution Protocol by dApp Clients

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

For usability, HyperService imposes smaller requirements on the watching daemons implemented by $\mathsf{Prot}_{\mathsf{CLI}}$. Specially, $\mathsf{Prot}_{\mathsf{CLI}}$ still proactively watches $\mathsf{Prot}_{\mathsf{NSB}}$ to have a fallback communication medium with $\mathsf{Prot}_{\mathsf{VES}}$. However, $\mathsf{Prot}_{\mathsf{CLI}}$ 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 $\mathsf{Prot}_{\mathsf{VES}}$ to enable lightweight dApp clients. $\mathsf{Prot}_{\mathsf{CLI}}$, 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 $\mathsf{Prot}_{\mathsf{ISC}}$. It generates the arbitration code, denoted as *contract*, based on the given dApp executable $\mathcal{G}_T$. The *contract* internally uses $\mathsf{T}_{\mathsf{state}}$ to track the state of each transaction in $\mathcal{G}_T$, 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 $\mathsf{st}_{\mathsf{proof}}$ and $\mathsf{A}_{\mathsf{revs}}$. When the $\mathsf{Prot}_{\mathsf{ISC}}$ times out, it executes the contract terms based on its internal state, after which its funds are depleted and the contract never runs again. Below we explain several technical subtleties.

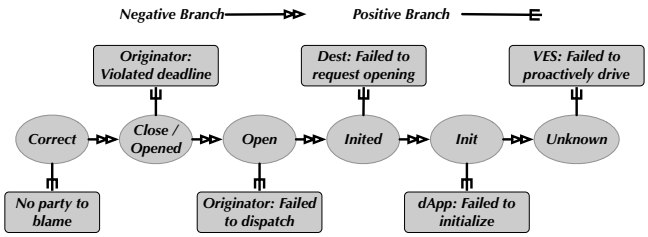

Figure 9: The decision tree to decide the accountable party for a dirty transaction.

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 $\text{Merk}^{i}_{\mathcal{T}}$, $\text{Merk}^{\text{id}}_{\mathcal{T}}$ or $\text{Merk}^{o}_{\mathcal{T}}$, Prot_{ISC} retrieves the single-party signed certificate $\text{Cert}^{i}_{\mathcal{T}}$, $\text{Cert}^{\text{id}}_{\mathcal{T}}$ or $\text{Cert}^{o}_{\mathcal{T}}$ 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.,* $\text{Cert}^{i}_{\mathcal{T}}$ is signed by Prot_{VES}, $\text{Cert}^{\text{id}}_{\mathcal{T}}$ is signed by $\mathcal{T}$'s originator and $\text{Cert}^{o}_{\mathcal{T}}$ is signed by the destination of $\mathcal{T}$. (ii) The enclosed on-chain transaction $\widetilde{T}$ in $\text{Cert}^{\text{id}}_{\mathcal{T}}$ and $\text{Cert}^{o}_{\mathcal{T}}$ 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 $\text{Cert}^{o}_{\mathcal{T}}$ is genuine, where the genuineness is defined as a bounded difference between ts_{open} and the height of the NSB block that attaches $\text{Merk}^{o}_{\mathcal{T}}$.

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 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 $\mathcal{G}_T$, which are the transactions that are eligible to be opened, but with non-correct state. Thus, the execution succeeds only if $\mathcal{G}_T$ 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 $\text{Cert}^{\text{id}}_{\mathcal{T}}$ 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 $\text{Cert}^{i}_{\mathcal{T}}$ 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 Specification of Prot_{NSB} and Prot_{BC}

Prot_{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 Prot_{BC} as a conceptual party trusted for correctness and availability, *i.e.,* Prot_{BC} guarantees to correctly perform any predefined computation (*e.g.,* Turing-complete smart contract programs) and is always available to handle user requests despite unbounded response latency. Prot_{NSB} specifies the protocol realization of the NSB. Prot_{NSB} is an extended version of Prot_{BC} with additional capabilities. Due to space constraint, we move the detailed protocol description of Prot_{BC} and Prot_{NSB} to our technical report that is available on both our source code repository [4] and arXiv.

4.6 Security Theorems

To rigorously prove the security properties of UIP, we first present the cryptography abstraction of the UIP in form of an ideal functionality $\mathcal{F}_{\text{UIP}}$. The ideal functionality articulates the correctness and security properties that UIP wishes to attain by assuming a trusted entity. Then we prove that Prot_{UIP}, our the decentralized real-world protocol containing the aforementioned preliminary protocols, securely realizes $\mathcal{F}_{\text{UIP}}$ using the UC framework [28], *i.e.,* Prot_{UIP} achieves the same functionality and security properties as $\mathcal{F}_{\text{UIP}}$ without assuming any trusted authorities. Since the rigorous proof requires non-trivial simulator construction within the UC framework, we defer detailed proof to a dedicated section § 8.

5 IMPLEMENTATION AND EXPERIMENTS

In this section, we present the implementation of a HyperService prototype and report experiment results on the prototype. At the time of writing, the total development effort includes (i) ~1,500 lines of Java code and ~3,100 lines of ANTLR [54] grammar code for building the HSL programming framework, (ii) ~21,000 lines of code, mainly in Go and Python, for implementing the UIP protocol; and ~8,000 lines of code, mainly in Go, for implementing the NSB; and (iii) ~1,000 lines of code, in Solidity, Vyper, Go and HSL, for writing cross-chain dApps running on HyperService. The released source code is available at [4]. The HyperService Consortium is under active development for HyperService.

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 [17] consensus engine, a commonly cited cornerstone for building enterprise blockchains. We implement the necessary accounts (wallets), the smart contract environment, and the on-chain storage to deliver 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 front end and the HSL front end using ANTLR [54], which parse the input HSL program and contracts, build an 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 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*: it requires no protocol change to both UIP and the blockchain itself. We simply need to add an extra parser to the multi-lang front end to support the programming language used by the blockchain (if this language is new to HyperService), and meanwhile VESes extend their visibility to this blockchain. The HyperService consortium is continuously working on on-boarding additional blockchains, both permissioned and permissionless.

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 [11]), or using trusted hardware enclaves [62]. HyperService, for the first time, realizes the possibility of *using blockchains themselves as oracles*. With the built-in decentralization and correctness guarantees 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 natively traded on different blockchains (a scaled-up version of the introductory example 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 [9], an initial coin offering in which tokens can be sold in various cryptocurrencies, and a gaming platform where players can freely trade and redeem their valuables (in form of non-fungible tokens) 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 regions can submit their votes to their regional blockchains, and the logic for computing the final votes based on the regional votes is publicly visible on another blockchain.

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 purposes, dApp clients and VES nodes can be deployed either locally or on cloud.

	Financial Derivatives		CryptoAsset Movement		Federated Computing	
	Mean	%	Mean	%	Mean	%
HSL Compilation	1.1769	~16	0.2598	~4	1.095	~15
Session Creation	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.

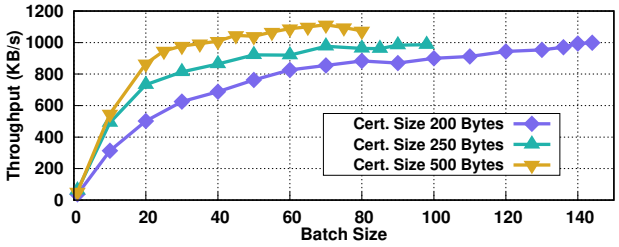

Figure 10: The throughput of the NSB, measured as the total size of committed certificates on the NSB per second.

5.3.1 End-to-End Latency

We evaluated all three applications mentioned in § 5.2 and reported their end-to-end execution latency introduced by HyperService in Table 3. The reported latency includes HSL program compiling, dApp-VES session creation, and (batched) NSB action staking and proof retrieval during the 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 claims in the end-to-end latency because they can be done offline anytime before the ISC expires.

These dApps show similar latency profiling breakdown, where the session creation 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 a 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 different since their executables contain different number of transactions.

5.3.2 NSB Throughput and HyperService Capacity

The throughput of the NSB affects aggregated dApp capacity on HyperService. In this section, we report the peak throughput of the currently implemented NSB. We stress tested the NSB by initiating up to one thousand dApp clients and VES nodes, which concurrently dispatched action and status staking to the NSB. We batched multiple certificate stakings by different clients into a single

NSB-transaction, so that the effective certificate-staking throughput perceived by those clients can exceed the consensus limit of the NSB. Figure 10 plots the NSB throughput, measured as the total size of committed certificates by all clients per second, under different certificate and batch sizes. The results show that as the batch size increases, regardless of the certificate sizes, the NSB throughput converged to about 1000 kilobytes per second. Given any certificate size, further enlarging the batch size cannot boost the throughput, whereas the failure rate of certificate staking increases, indicating that the NSB is fully loaded.

Given the above NSB throughput, the actual dApp capacity of the HyperService platform further depends on how often the communication between dApp clients and VESes falls back to the NSB. In particular, each dApp-transaction spawns at most six NSB-transactions (five action stakings and one status staking), assuming that the off-chain channel is fully nonfunctional (zero NSB transaction if otherwise). Thus, the *lower bound* of the aggregate dApp capacity on HyperService, which would be reached only if all off-chain channels among dApp clients and VESes were simultaneously broken, is about $\frac{170000}{s}$ transactions per second (TPS), where s is the (average) size (in bytes) of a certificate. This capacity and the TPS of most PoS production blockchains are of the same magnitude. Further, considering (i) the NSB is horizontally shardable at the granularity of each underlying blockchain (§ 4.1.3) and (ii) not all transactions on an underlying blockchain are cross-chain related, we anticipate that the NSB will not become the bottleneck as HyperService scales to support more blockchains in the future.

6 DISCUSSION

In this section, we discuss several aspects that have not been thoroughly addressed in this paper, and present our vision for future work on HyperService and its impact.

6.1 Programming Framework Extension

HSL is a high-level programming language designed to write cross-chain dApps under the USM programming model. The language constructs provided by HSL allow developers to directly specify entities, operations, and dependencies in HSL programs. To ensure the determinism of operations, which is an important property for the NSB and the ISC to determine the correctness or violation of dApp executions, the language constructs do not include control-flow operations such as conditional branching, looping, and calling/returning from a procedure. Additionally, dynamic transaction generation is also not supported by HSL, since it has led to a new class of bugs known as re-entrancy vulnerabilities [55]. These design choices are consistent with the recent blockchain programming languages that emphasize on safety guarantees, such as Move [22] for Facebook's Libra blockchain.

In future work, we plan to extend the design of the UIP protocol to support *dynamic transaction graphs*, which allows conditional execution of operations and certain degree of indeterminism of operation executions, such as repeating an operation for a specific times based on the values of state variables computed from previous operations. With those extensions, we are able to implement control-flow operations into HSL and provide both static and dynamic verification to ensure the correctness of dApps.

6.2 Cross-Shards and Cross-Worlds

HyperService is motivated by heterogeneous blockchain interoperation. Thanks to its generic design, HyperService can also enable cross-shard smart contracting and transactions for sharded blockchain platforms (*e.g.,* OmniLedger [44] and RapidChain [60]). On the one hand, the HSL programming framework is blockchain-neutral and extensible. Thus, writing dApps that involve smart contracts and accounts on different blockchains is conceptually identical to writing dApps that operate contracts and accounts on different shards. In fact, given that most of those sharded blockchains are homogeneously sharded (*i.e.,* all shards have the same format of contracts and accounts), developing and compiling cross-shard dApps using HSL are even simpler than cross-chain dApps. On the other hand, realizing UIP on sharded blockchains also requires less overhead since maintaining an NSB for all (homogeneous) shards is more lightweight than maintaining an NSB supporting heterogeneous blockchains. In fact, many sharded blockchain platforms already maintain a dedicated global blockchain as their trust anchor (*e.g.,* the identity chain of OmniLedger [44] and the beacon chain of Harmony [2]), to which the NSB functionality can be ported.

Additionally, we envision that the fully connected Web 3.0 should also include centralized platforms (*i.e.,* Cloud) to compensate for functionality (*e.g.,* performing computationally intensive tasks) that is difficult to execute on-chain. We recognize that two additional capabilities, with minimal distribution of their operation models, are required from those centralized platforms to make them compatible with HyperService: (i) any public state they publish should be coupled with verifiable proofs to certify the correctness of the state (where the definition of correctness could be application-specific), and (ii) all published state should have the concept of finality. With such capabilities, dApps on HyperService can trustlessly incorporate the state published by those centralized platforms.

6.3 Interoperability Service Providers

VESes play vital roles on HyperService platform. We envision that VESes would enter the HyperService ecosystem as Cross-chain Interoperability Service Providers (CSPs) by providing required services to support cross-chain dApps, such as compiling HSL programs into transaction dependency graphs and speaking the UIP protocol. This vision is indeed strengthened by the practical architectures of production blockchains, where all peer-to-peer nodes evolve into a hierarchy of stakeholders and a number of organizations operate (without necessarily owning) most of the mining power for Proof-of-Work blockchains or/and stakes for Proof-of-Stake blockchains (whether such a hierarchical architecture undermines decentralization is debatable, and beyond the scope of HyperService). Those organizations are perfectly qualified to operate as CSPs since they have good connectivity to multiple blockchains and maintain sufficient token liquidity to support insurance staking, contract invocation, and token transfers that are required in a wide range of cross-chain dApps.

CSPs (VESes) could be found via a community-driven *directory* (similar to Tor's relay directories [18, 31]), which we envision to be an informal list of CSPs. Each CSP has its own operation models, including the set of reachable blockchains, service fees charged for correct dApp executions, and insurance plans to compensate

for CSP-induced dApp failures. Developers have full autonomy to select CSPs based on their dApp requirements. Since all dApp execution results are publicly verifiable, it is possible to build a CSP reputation system to provide a valuable metric for CSP selection. CSPs thus misbehave at their own risk.

Because a CSP may wish to limit its staked funds at risk in the ISC, a dApp may be too large for any single CSP. Alternatively, a dApp may span a set of blockchains such that no single CSP has reachability to all of them. In such cases, a cross-chain dApp could be co-executed by a collection of VESes. By design, HyperService allows multi-VES executions since the UIP protocol does not restrict the number of VESes or dApp clients.

We envision the industrial impact of HyperService to be the birth of a CSP-formed *liquidity network* interconnected by the UIP protocol, powering a wide range of cross-chain dApps.

6.4 Complete Atomicity for dApps

In the context of cross-chain applications, dApps should be treated as first-class citizens because the success or failure of any individual transaction cannot fully decide the state of a dApp. HyperService follows this design philosophy by providing security guarantees at the granularity of dApps. However, the current version of HyperService is not fully dApp-atomic since UIP is unable to revert any state update to smart contracts when a dApp terminates prematurely. We recognize this as a fundamental challenge due to the finality guarantee of blockchains.

To deliver full dApp-atomicity on HyperService, we propose the concept of *stateless smart contracts* where contracts are able to their load state from the blockchain before execution. As a result, even if the state persistent on block $\mathcal{B}_n$ for a smart contact C eventually becomes dirty due to dApp failure, subsequent dApps can still load clean state for the contract C from a block (prior to $\mathcal{B}_n$) agreed by all parties. Although this design imposes additional requirements on underlying blockchains, it is practical and deliverable using "layer-two" protocols where smart contract executions could be decoupled from the consensus layer, for instance, via the usage of Trusted Execution Environment (*e.g.,* Intel SGX [30] and Keystone [48]).

6.5 Privacy-Preserving Blockchains

The primary challenge of supporting privacy-preserving blockchains on HyperService is the lack of a generic abstraction for those systems. In particular, various designs have been proposed to enhance blockchain privacy, such as encrypting blockchain state [29], obfuscating and mixing transactions via cryptography signature [58]. As a result, none of those blockchains can be abstracted as generic programmable state machines. Therefore, our approach towards interoperating privacy-preserving blockchains will be dApp-specific, such as relying on fast zero-knowledge proofs [25] to allow dApps to certify the state extracted from those blockchains.

7 RELATED WORK

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 [21, 36, 41], atomic cross-chain swaps [5, 38], and cryptocurrency-backed assets [61]. However, programmability, *i.e.,* smart contracting across heterogeneous blockchains, is largely ignored in those protocols.

In industry, Cosmos [7] and Polkadot [12] 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 [17] for Cosmos and Substrate [16] 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", we also notice two notable differences between them and HyperService. First and foremost, the cross-chain layer of Cosmos, powered by its Inter-blockchain Communication Protocol (IBC) [15], mainly focuses on preliminary network-level communications. In contrast, HyperService proposes a complete stack of designs with a unified programming framework for writing cross-chain dApps and a provably secure cryptography protocol to execute dApps. Further, at the time of writing, the most recent development of Cosmos and industry adoption are heading towards *homogeneity* where only Tendermint-powered blockchains are interoperable [1]. This is in fundamental contrast with HyperService where the blockchain heterogeneity is a first-class design requirement. Polkadot proceeds relatively slower than Cosmos: Substrate is still in early stage [16].

Existing blockchain platforms such as Ethereum [59] and Nebulas [10] allow developers to write contracts using new languages such as Solidity [14] and Vyper [19] or a tailored version of the existing languages such as Go, Javascript, and C++. Facebook recently released Move [22], a programming language in their blockchain platform Libra, which adopts the move semantics of Rust and C++ to prohibit copying and implicitly discarding coins and allow only move of the coins. To unify these heterogeneous programming languages, we propose HSL that 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 security-related fields that have a well-established corpus of low level algorithms, such as secure overlay networks [42, 49], network intrusions [23, 56, 57], and enterprise systems [33, 34], these languages are explicitly designed to solve their domain-specific problems, and cannot meet the needs of the unified programming framework for writing cross-chain dApps.

8 SECURITY THEOREMS

In this section, we present the main security theorems for UIP, and rigorously prove them using the UC-framework [28].

8.1 Ideal Functionality $\mathcal{F}_{\mathsf{UIP}}$

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, $\mathcal{F}_{\mathsf{UIP}}$ generates keys for both

Init: Data := ∅
Upon Receive SessionCreate($\mathcal{G}_T$, *contract*, $\mathcal{P}_a$, $\mathcal{P}_z$):
  generate the session ID sid ← $\{0,1\}^\lambda$ and keys for both parties
  send Cert([sid, $\mathcal{G}_T$, *contract*]; $\text{Sig}_{\text{sid}}^{\mathcal{P}_z}$, $\text{Sig}_{\text{sid}}^{\mathcal{P}_a}$) 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 $\text{Cert}_{\mathcal{T}}^{i}$:= Cert([$\mathcal{T}$, init, sid]; $\text{Sig}_{\text{sid}}^{\mathcal{P}_z}$)
  send $\text{Cert}_{\mathcal{T}}^{i}$ 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 $\text{Cert}_{\mathcal{T}}^{\text{id}}$:= Cert([$\widetilde{T}$, inited, sid, $\mathcal{T}$]; $\text{Sig}_{\text{sid}}^{\mathcal{P}}$)
  send $\text{Cert}_{\mathcal{T}}^{\text{id}}$ 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 ts_{open} := now()
  compute $\text{Cert}_{\mathcal{T}}^{o}$:= Cert([$\widetilde{T}$, open, ts_{open}, sid, $\mathcal{T}$]; $\text{Sig}_{\text{sid}}^{\mathcal{P}}$)
  send $\text{Cert}_{\mathcal{T}}^{o}$ to both $\{\mathcal{P}_a, \mathcal{P}_z\}$ to inform action
Upon Receive ReqTransOpened($\mathcal{T}$, sid, $\widetilde{T}$, $\mathcal{P}$, ts_{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 ts_{open} is within the error boundary with now()
  update $\mathcal{T}$.state = opened and get $\mathcal{T}.\text{ts}_{\text{open}}$:= ts_{open}
  post $\widetilde{T}$ on $\mathcal{F}_{\text{blockchain}}$ for on-chain execution
  compute $\text{Cert}_{\mathcal{T}}^{\text{od}}$:= Cert([$\widetilde{T}$, open, ts_{open}, sid, $\mathcal{T}$]; $\text{Sig}_{\text{sid}}^{\mathcal{P}_a}$, $\text{Sig}_{\text{sid}}^{\mathcal{P}_z}$)
  send $\text{Cert}_{\mathcal{T}}^{\text{od}}$ to both $\{\mathcal{P}_a, \mathcal{P}_z\}$ to inform action
Upon Receive ReqTransClose($\mathcal{T}$, sid, $\widetilde{T}$, $\text{ts}_{\text{closed}}$):
  $(\mathcal{G}_T, \_, \_, \_)$:= Data[sid]; abort if not found
  assert $\mathcal{T}$.state = opened and $\mathcal{T}$.trans = $\widetilde{T}$
  query the ledger of $\mathcal{F}_{\text{blockchain}}$ for $\widetilde{T}$'s status
  abort if T is not finalized on $\mathcal{F}_{\text{blockchain}}$
  assert $\text{ts}_{\text{closed}}$ is within the error boundary with current time now()
  update $\mathcal{T}$.state := closed and $\mathcal{T}.\text{ts}_{\text{closed}}$:= $\text{ts}_{\text{closed}}$
  compute $\text{Cert}_{\mathcal{T}}^{c}$:= Cert([T, closed, $\text{ts}_{\text{closed}}$, sid, $\mathcal{T}$]; $\text{Sig}_{\text{sid}}^{\mathcal{P}_a}$, $\text{Sig}_{\text{sid}}^{\mathcal{P}_z}$)
  send $\text{Cert}_{\mathcal{T}}^{c}$ 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 $\mathcal{F}_{\text{UIP}}$.

parties, allowing $\mathcal{F}_{\text{UIP}}$ to sign transactions and compute certificates on their behalf. Both parties are required to stake sufficient funds, derived from the *contract*, into $\mathcal{F}_{\text{UIP}}$. $\mathcal{F}_{\text{UIP}}$ annotates each transaction 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 $\mathcal{F}_{\text{UIP}}$ with the real-world protocol Prot_{UIP}, in Figure 11, we assume that $\mathcal{P}_a$ is the dApp client and $\mathcal{P}_z$ is the VES.

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

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

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

Verbose Definition of $\mathcal{F}_{\text{UIP}}$. We *intentionally* define $\mathcal{F}_{\text{UIP}}$ verbosely (that is, sending many signed messages) in order to accurately match $\mathcal{F}_{\text{UIP}}$ to the real world protocol Prot_{UIP}. For instance, in the SessionCreate interface, $\mathcal{F}_{\text{UIP}}$ certifies ($\mathcal{G}_T$, *contract*, sid) on behalf of both parties to simulate the result of a successful handshake between two parties in the real world. Another example is that the attestations generated in those state update interfaces are not essential to ensure correctness due to the assumed trustiness of $\mathcal{F}_{\text{UIP}}$. However, $\mathcal{F}_{\text{UIP}}$ still publishes attestations to emulate the *side effects* of Prot_{UIP} in the real world. As we shall see below, such emulation is crucial to prove that $\mathcal{F}_{\text{UIP}}$ UC-realizes Prot_{UIP}.

Correctness and Security Properties of $\mathcal{F}_{\text{UIP}}$. With the assumed trustiness, it is not hard to see that $\mathcal{F}_{\text{UIP}}$ 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 leave the support for privacy-preserving blockchains on HyperService to future work.

8.2 Main Security Theorems

In this section, we claim the main security theorem of HyperService. The correctness of Theorem 8.1 guarantees that $\mathsf{Prot}_{\mathsf{UIP}}$ achieves same security properties as $\mathcal{F}_{\mathsf{UIP}}$.

Theorem 8.1. *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 $\mathcal{F}_{UIP}$ against a malicious adversary in the passive corruption model.*

We further 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 their communication medium.

Theorem 8.2. *With the same assumption of Theorem 8.1, the UIP protocol variant $\mathsf{H\text{-}Prot}_{UIP}$ securely UC-realizes the ideal functionality $\mathcal{F}_{UIP}$ against a malicious adversary in the Byzantine corruption model.*

8.3 Proof Overview

We now the prove our main theorems. We start with Theorem 8.1. In the UC framework [28], the model of $\mathsf{Prot}_{\mathsf{UIP}}$ 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 $\mathsf{Prot}_{\mathsf{UIP}}$ where each party may execute different parts of $\mathsf{Prot}_{\mathsf{UIP}}$. Intuitively, the environment $\mathcal{E}$ represents the *external* system that contains other protocols, including ones that provide inputs to, and obtain outputs from, $\mathsf{Prot}_{\mathsf{UIP}}$. 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 8.1) 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. § 8.6 discusses the *Byzantine* corruption model, where the adversary takes complete control of the corrupted party.

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

The ideal protocol $\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$ is a wrapper around $\mathcal{F}_{\mathsf{UIP}}$ by a set of dummy parties that have the same interfaces as the parties of $\mathsf{Prot}_{\mathsf{UIP}}$ in the real world. As a result, $\mathcal{E}$ is able to interact with $\mathcal{I}_{\mathcal{F}_{\mathsf{UIP}}}$ in the ideal world the same way it interacts with $\mathsf{Prot}_{\mathsf{UIP}}$ in the real world. These dummy parties simply pass received input from $\mathcal{E}$ to $\mathcal{F}_{\mathsf{UIP}}$ and relay output of $\mathcal{F}_{\mathsf{UIP}}$ to $\mathcal{E}$, without implementing any additional logic. $\mathcal{F}_{\mathsf{UIP}}$ controls all keys of these dummy parties. For the sake of clear presentation, we abstract the real-world participants of $\mathsf{Prot}_{\mathsf{UIP}}$ as five parties $\{\mathcal{P}_{\mathsf{VES}}, \mathcal{P}_{\mathsf{CLI}}, \mathcal{P}_{\mathsf{ISC}}, \mathcal{P}_{\mathsf{NSB}}, \mathcal{P}_{\mathsf{BC}}\}$. 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 [28], 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.

8.4 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.,* § 8.5) and constructs a SessionCreate call to $\mathcal{F}_{\mathsf{UIP}}$ with parameter $(\mathcal{G}_T, \mathit{contract}, \mathcal{P}^{\mathcal{I}}_{\mathsf{CLI}}, \mathcal{P}^{\mathcal{I}}_{\mathsf{VES}})$.
- 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 $(\mathcal{T}, \mathsf{sid}, \mathcal{P}^{\mathcal{I}}_{\mathsf{VES}})$. 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 $\mathcal{P}^{\mathcal{I}}_{\mathsf{ISC}}$.

8.5 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 [28], 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 $\mathbf{A_1}$. 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 $(\mathcal{G}_T$, *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 § 8.4, $\mathcal{S}$ 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 $\mathcal{F}_{UIP}$, 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 $\mathbf{A_2}$. 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 all 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 $\mathbf{A_3}$. 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 $\mathbf{A_4}$, *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.

8.6 Byzantine Corruption Model

Theorem 8.1 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 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 public and 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 8.2.

9 CONCLUSION

In this paper, we presented HyperService, the first platform that offers interoperability and programmability across heterogeneous blockchains. HyperService is powered by two 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 approximately 35,000 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 aggregate platform throughput.

10 ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We thank Harmony Protocol for their discussion on cross-shard transactions. This material is based upon work partially supported by NSF under Contract Nos. CNS-1717313 and TWC-1518899, and by National Key Research and Development Program of China under grant No. 2018YFB0803605 and NSFC under grant No. 61702045. Correspondence authors are Zhuotao Liu and Haoyu Wang.

REFERENCES

[1] Cosmos WhitePaper. https://cosmos.network/resources/whitepaper, 2019.
[2] Harmony: Technical Whitepaper. https://harmony.one/whitepaper.pdf, 2019.
[3] Monoxide: Scale Out Blockchain with Asynchronized Consensus Zones. In *USENIX NSDI* (2019).
[4] Open Source Code for HyperService by HyperService-Consortium. https://github.com/HyperService-Consortium, 2019.
[5] Bitcoin Wiki: Atomic Cross-Chain Trading. https://en.bitcoin.it/wiki/Atomic_swap, Accessed on 2019.
[6] CoinMarketCap. https://coinmarketcap.com, Accessed on 2019.
[7] Cosmos. https://cosmos.network, Accessed on 2019.
[8] DPOS Consensus Algorithm. https://steemit.com/dpos/@dantheman/dpos-consensus-algorithm-this-missing-white-paper, Accessed on 2019.
[9] J.P. Morgan: Blockchain and Distributed Ledger. https://www.jpmorgan.com/global/blockchain, Accessed on 2019.
[10] Nebulas. https://github.com/nebulasio, Accessed on 2019.
[11] Oraclize. http://www.oraclize.it, Accessed on 2019.
[12] Polkadot. https://polkadot.network, Accessed on 2019.
[13] rhombus. https://rhombus.network, Accessed on 2019.
[14] Solidity. https://solidity.readthedocs.io/en/v0.5.6/, Accessed on 2019.
[15] Standards for the Cosmos network & Interchain Ecosystem. https://github.com/cosmos/ics, Accessed on 2019.
[16] Substrate. https://github.com/paritytech/substrate, Accessed on 2019.
[17] Tendermint Core. https://tendermint.com, Accessed on 2019.
[18] Tor Directory Authorities. https://metrics.torproject.org/rs.html#search/flag:authority, Accessed on 2019.
[19] Vyper. https://github.com/ethereum/vyper, Accessed on 2019.
[20] Al-Bassam, M., Sonnino, A., Bano, S., Hrycyszyn, D., and Danezis, G. Chainspace: A Sharded Smart Contracts Platform. *NDSS* (2017).
[21] 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: tinyurl. com/mj656p7* (2014).
[22] Blackshear, S., Cheng, E., Dill, D. L., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Rain, Russi, D., Sezer, S., Zakian, T., and Zhou, R. Move: A language with programmable resources. Tech. rep., The Libra Association, 2019.
[23] Borders, K., Springer, J., and Burnside, M. Chimera: A Declarative Language for Streaming Network Traffic Analysis. In *USENIX Security Symposium* (2012).
[24] 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).
[25] Bünz, B., Bootle, J., Boneh, D., Poelstra, A., Wuille, P., and Maxwell, G. Bulletproofs: Short proofs for Confidential Transactions and More. In *2018 IEEE Symposium on Security and Privacy (SP)* (2018), IEEE, pp. 315–334.
[26] Buterin, V. Chain Interoperability. *R3 Reports* (2016).
[27] Buterin, V., et al. A Next-Generation Smart Contract and Decentralized Application Platform. *white paper* (2014).
[28] Canetti, R. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *IEEE Symposium on Foundations of Computer Science* (2001).
[29] Cheng, R., Zhang, F., Kos, J., He, W., Hynes, N., Johnson, N. M., Juels, A., Miller, A., and Song, D. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution. In *IEEE EuroS&P* (2019).
[30] Costan, V., and Devadas, S. Intel SGX explained, Accessed on 2019. https://eprint.iacr.org/2016/086.pdf.
[31] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium* (2004).
[32] Eyal, I., Gencer, A. E., Sirer, E. G., and Van Renesse, R. Bitcoin-NG: A Scalable Blockchain Protocol. In *USENIX NSDI* (2016).
[33] Gao, P., Xiao, X., Li, D., Li, Z., Jee, K., Wu, Z., Kim, C. H., Kulkarni, S. R., and Mittal, P. SAQL: A Stream-based Query System for Real-time Abnormal System Behavior Detection. In *USENIX Security Symposium* (2018).
[34] Gao, P., Xiao, X., Li, Z., Jee, K., Xu, F., Kulkarni, S. R., and Mittal, P. AIQL: Enabling Efficient Attack Investigation from System Monitoring Data. In *USENIX ATC* (2018).
[35] Garay, J., Kiayias, A., and Leonardos, N. The Bitcoin Backbone Protocol with Chains of Variable Difficulty. In *Annual International Cryptology Conference* (2017).
[36] Gazi, P., Kiayias, A., and Zindros, D. Proof-of-stake Sidechains. In *IEEE Symposium on Security & Privacy* (2019).
[37] Green, M., and Miers, I. Bolt: Anonymous Payment Channels for Decentralized Currencies. In *ACM CCS* (2017).
[38] Herlihy, M. Atomic Cross-Chain Swaps. In *ACM PODC* (2018).
[39] Kalodner, H., Goldfeder, S., Chen, X., Weinberg, S. M., and Felten, E. W. Arbitrum: Scalable, Private Smart Contracts. In *USENIX Security Symposium* (2018).
[40] Khalil, R., and Gervais, A. Revive: Rebalancing Off-blockchain Payment Networks. In *ACM CCS* (2017).
[41] Kiayias, A., and Zindros, D. Proof-of-work Sidechains. Tech. rep., Cryptology ePrint Archive, Report 2018/1048, 2018.
[42] Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. Mace: Language support for building distributed systems. In *ACM PLDI* (2007).
[43] 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).
[44] 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* (2018).
[45] 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* (2016).
[46] Krupp, J., and Rossow, C. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *USENIX Security Symposium* (2018).
[47] Lamport, L. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM* (1978).
[48] Lee, D., Kohlbrenner, D., Shinde, S., Song, D., and Asanović, K. Keystone: A Framework for Architecting TEEs. *arXiv preprint arXiv:1907.10119* (2019).
[49] 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).
[50] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. Making Smart Contracts Smarter. In *ACM CCS* (2016).
[51] Luu, L., Narayanan, V., Zheng, C., Baweja, K., Gilbert, S., and Saxena, P. A Secure Sharding Protocol for Open Blockchains. In *ACM CCS* (2016).
[52] Malavolta, G., Moreno-Sanchez, P., Kate, A., Maffei, M., and Ravi, S. Concurrency and Privacy with Payment-channel Networks. In *ACM CCS* (2017).
[53] Nakamoto, S. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, 2008.
[54] Parr, T. Antlr. https://www.antlr.org/, 2014.
[55] Sergey, I., and Hobor, A. A Concurrent Perspective on Smart Contracts. In *Financial Cryptography and Data Security* (2017).
[56] Sommer, R., Vallentin, M., De Carli, L., and Paxson, V. Hilti: An abstract execution environment for deep, stateful network traffic analysis. In *IMC* (2014).
[57] Vallentin, M., Paxson, V., and Sommer, R. VAST: A Unified Platform for Interactive Network Forensics. In *USENIX NSDI* (2016).
[58] Van Saberhagen, N. CryptoNote v 2.0. https://cryptonote.org/whitepaper.pdf, 2013.
[59] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* (2014).
[60] Zamani, M., Movahedi, M., and Raykova, M. RapidChain: Scaling Blockchain via Full Sharding. In *ACM CCS* (2018).
[61] Zamyatin, A., Harz, D., Lind, J., Panayiotou, P., Gervais, A., and Knottenbelt, W. XCLAIM: Trustless, Interoperable, Cryptocurrency-Backed Assets. In *IEEE Symposium on Security and Privacy* (2019).
[62] Zhang, F., Cecchetti, E., Croman, K., Juels, A., and Shi, E. Town Crier: An Authenticated Data Feed for Smart Contracts. In *ACM CCS* (2016).