Make Web3.0 Connected

A Perspective from Interoperability and Programmability across Blockchains

Zhuotao Liu, Yangxi Xiang, Jian Shi, Peng Gao, Haoyu Wang, Xusheng Xiao, Bihan Wen, Qi Li, and Yih-Chun Hu

Abstract— Web3.0, often cited to drastically shape our lives, is ubiquitous. However, few literatures have discussed the crucial differentiators that separate Web3.0 from the era we are currently living in. Via a thorough analysis of the recent blockchain infrastructure evolution, we capture a key invariant featuring the evolution, based on which we provide the first academic definition for Web3.0. Our definition is not the only way of understanding Web3.0, yet, it captures the fundamental and defining trait of Web3.0, and meanwhile it is has two desirable properties. Under this definition, we articulate three key categories of infrastructural enablers for Web3.0: individual smart-contract capable blockchains, federated or centralized platforms capable of publishing verifiable states, and an interoperability platform to hyperconnect those state publishers to provide a unified and connected computing platform for Web3.0 applications. While innovations in all categories are necessary to fully enable Web3.0, in this paper, we present a design for the third enabler, *i.e.*, the first interoperability platform, namely HyperService, that advances the state-of-the-art by simultaneously delivers *interoperability* and *programmability* across *heterogeneous* blockchains and state publishers. 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 62,000 lines of code to demonstrate its practicality, usability and scalability.

1 INTRODUCTION

In the past few years, the keyword *Web3.0* is ubiquitous, driven by the hyper-enthusiasm for cryptocurrency and Blockchain. Although Web3.0 is often cited to drastically shape our lives, few literatures have discussed the crucial differentiators that separate Web3.0 from the era we are currently living in. As a result, our perception of Web3.0 is still preliminary, obfuscated by a list of endless fancy words and terms such as cryptocurrency, Bitcoin, blockchain, decentralization, ICOs, anti-monopoly, data-ownership, "software is eating law", etc.

Defining Web3.0 meaningfully is non-trivial. Over the past few years, we had numerous conversations and interviews with experts and practitioners in various Web3.0-related industry sectors, including layer-one blockchain infrastructures, layer-two (off-chain) protocols, consortium (or enterprise) blockchain service providers, and decentralized applications (*e.g.*, decentralized finance and gaming), hoping to understand (i) what are the fundamental and defining characteristics of Web3.0 and (ii) what are the key enablers of Web3.0. We observed that although they all claim their infrastructures or/and applications are indeed Web3.0 products by enumerating various advantages over the so-called Web2.0 counterparts, it is challenging to abstract away their product-specific buzzwords to reach a crystal-clear Web3.0 definition, letting alone articulating the key enablers of Web3.0.

In this paper, we lay out our observations about Web3.0 (partially shaped by the aforementioned industrial study) and the reasoning on those findings, to provide the first academic definition for Web3.0. We do not claim that our definition is the only way of understanding Web3.0, yet it has two desirable properties: generic and measurable. It is generic since it is not limited to any overarching applications or underlying infrastructures; it is measurable because all stakeholders can determine an application's eligibility for the Web3.0 era using a fundamental and defining trait of Web3.0 that we captured via a thorough analysis of the recent blockchain infrastructure evolution. Under this Web3.0 definition, we articulate three concrete key infrastructural enablers for Web3.0: (i) individual blockchains with enhanced performance and security properties to serve as the ideal platforms to support verifiable computing; (ii) federated or centralized platforms, capable of publishing verifiable states, to compensate for the functionality that is difficult or infeasible to realize on-chain; and (iii) a secure interoperability platform to hyperconnect these distributed and isolated state publishers (*i.e.*, both blockchains and federated / centralized platforms) to provide a unified and connected computing platform for Web3.0 applications.

While innovations in all three key enablers are necessary to fully enable Web3.0, in this paper, we present in detail a design for the third key enabler, *i.e.*, the first interoperability platform for Web3.0. Throughout the paper, we describe our protocols mostly in the context of interoperating heterogeneous blockchains and extend interoperability to include federated or centralized state publishers in § 5.7. Existing interoperability proposals [2–5] mostly focus on atomic token exchange between two blockchains without centralized exchanges. However, since smart contracts executing on blockchains have transformed blockchains from append-only distributed ledgers into programmable state machines, *token exchange is not the complete scope of blockchain interoperability*. Instead, blockchain interoperability is complete only with *programmability*, allowing developers to write Web3.0 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 Web3.0 decentralized applications (or dApps) is unclear. In general, from

- *Z. Liu is with Google Inc. and University of Illinois at Urbana-Champaign, USA.*
- *Y. Xiang and H. Wang are with Beijing University of Posts and Telecommunications, China.*
- *J. Shi and X. Xiao are with Case Western Reserve University, USA.*
- *P. Gao is with University of California, Berkeley, USA.*
- *B. Wen is with Nanyang Technological University, Singapore.*
- *Q. Li is with Tsinghua University, China.*
- *Y. Hu is with University of Illinois at Urbana-Champaign, USA.*
- *Part of the material in this manuscript appeared in [1].*

developers' perspective, it is desirable that cross-chain dApps could preserve the same state-machine-based programming abstraction as single-chain contracts [6]. 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 dApps.

Second, existing token-exchange oriented interoperability protocols, such as atomic cross-chain swaps (ACCS) [7], 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 § 3.3 invokes a smart contract using parameters obtained from smart contracts deployed on different blockchains, and meanwhile the condition of an invocation may even depend on the dynamic state on remote blockchains. The complexity of this operation is far beyond mere token transfers.

To meet these challenges, we propose HyperService, the first interoperability platform for building and executing Web3.0 dApps across heterogeneous blockchains. At the highest 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. UIP (short for universal inter-blockchain protocol) is our cryptography protocol that handles the complexity of cross-chain execution of dApps written in HSL. 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.

To summarize, we make the following key contributions.

(i) We provide the first generic and measurable metric to define the era of Web3.0, based on our observations and reasoning about the evolution of blockchain infrastructures over the past few years. Based on this definition, we articulate three key infrastructural enablers for Web3.0.

(ii) We present in detail one of the three key enablers: the first interoperability platform, HyperService, to hyperconnect heterogeneous blockchains and federated / centralized state publishers to provide a unified and connected computing platform for Web3.0 applications. We implement a prototype of HyperService in approximately 62,000 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 is horizontally scalable.

2 WHAT IS Web3.0 and WHAT DOES IT TAKE

In this section, we elaborate on our thoughts about Web3.0, focusing on answering the following two questions: (i) what are the fundamental and defining characteristics of Web3.0 and (ii) what are the key enablers of Web3.0.

Our high-level reasoning process is as follows. While Web3.0 is ubiquitous in industry sectors, there is surprisingly little academic work to systematically study it. Thus, we extend our study about Web3.0 by initiating numerous interviews and conversations with many experts and practitioners from a wide range of Web3.0-related industry sectors, including layer-one blockchain infrastructures, layer-two (off-chain) protocols, consortium (or enterprise) blockchain service providers, decentralized applications, etc. Then, based on a thorough analysis of the observed blockchain infrastructure evolution (see § 2.1), we distill a key invariant featuring the evolution, which is *verifiability* rather than "decentralization" or "trustlessness" or any other buzzwords that people have used to describe Web3.0. This invariant is somewhat surprising, yet it captures the most defining trait of the Web3.0 evolution. Third, centering around the key variant, we propose the first academic definition of Web3.0. Although our definition is not the only way of understanding Web3.0, it has two desirable properties as explained in § 2.3. Finally, under this definition, we propose three key areas of infrastructural innovations required to enable Web3.0. We summarize the academic research in each area and focus on presenting in detail a design for one of the key enablers.

2.1 Decentralization or Not

According to conventional wisdom, decentralization is almost the synonym for Web3.0. Admittedly, the enthusiasm for Web3.0 was ignited by the decentralized peer-to-peer payment network Bitcoin. Interestingly, the definition and realization of decentralization have evolved significantly over time. Nowadays, the mining power of Bitcoin concentrates on several large mining pools, which essentially undermines peer equality, one crucial attribute of the classic decentralization definition. Meanwhile, the emerging blockchains using (Delegated) Proof-of-Stake as their consensus protocols (due to various benefits including higher transaction throughput, faster transaction finality and easier governance) are inherently hierarchical, which, arguably, is more similar to the era we are currently living in than an ideal decentralized era.

The blockchain infrastructure continues to evolve as additional features are proposed by new projects, often cited as their market differentiators. Privacy-preserving smart contracting is one of the most frequently mentioned and highly desired features. The realization of privacy-preservation often relies on hardware primitives such as Trusted Execution Environment (TEE), because it is either infeasible or prohibitively expensive to generalize pure software-based solutions (often based on cryptography innovations, such as zero-knowledge proofs and secure multiparty computation) to support generic privacy-preserving computation in smart contracts. Paradoxically, Intel is the dominant TEE provider world wide. Another example is off-chain data feed (*i.e.,* oracle) which is essential to provide authentic external data for on-chain smart contracts (*e.g.,* decentralized finance, or DeFi, applications often need data from certified financial organizations). The authenticity of these data feeds, again, are not decided in a decentralized manner.

The ever-changing infrastructure realization of the decentralized blockchains probably means that *we should not equalize Web3.0 and decentralization*. Instead, despite the infrastructure evolvement, we observed that Web3.0 applications, in general, hold a key invariant: stakeholders are able to prove whether the execution of a Web3.0 application complies with or violates the pre-agreed contractual terms between users and the application. This invariant naturally holds when Web3.0 applications fully execute on-chain without external dependencies, regardless of the governance model of the underlying blockchains. Further, even when off-chain components are added, such as TEEs and oracles, they often provide security primitives to allow users to attest the correctness of code execution or the authenticity of data feeds. Thus, compared with decentralization, *verifiability* is a more precise defining-feature of Web3.0.

We also thought about several other candidate terms that may capture the aforementioned invariant, such as accountability, transparency, trustlessness and statefulness. However, accountability and statefulness are typically protocol-level features; Web3.0 applications are not necessarily transparent for privacy concerns; and although the Trusted Computing Base (TCB) for Web3.0 applications is generally small, it is still challenging to claim that they are trustless.

2.2 Critical Computing

As the complexity of Web3.0 applications continues to grow, executing a Web3.0 application completely on-chain or inside a TEE becomes increasingly challenging or even infeasible, because of the memory and computation capability limitations of the blockchains and TEEs. This naturally raises a question: what is the proper demarcation point for verifiability when an application requires non-trivial off-chain participation. Clearly, the demarcation point should be application-specific. To abstract it, we introduce the concept of *critical computing* which represents the key contract between users and the application that is deemed crucial to be verifiable. The critical computing is not necessarily the most complex, or computation-intensive, or even proprietary part of the application. Instead, it should define the high-level business commitment and agreement between users and the application. For instance, in various crypto-pet applications, the key contract should be how users' pets will evolve after gaining experiences or points by eating crypto-foods or winning battles; in DeFi applications, verifying that the financial terms (*e.g.,* the interest rates of lending contracts) are executed correctly is desirable.

Another key reason for explicitly defining critical computing for Web3.0 applications is the increasingly clear necessity of embracing federated or centralized computation platforms (*e.g.,* Cloud) in Web3.0. Going beyond preliminary applications such as ICOs and crypto-pets, Web3.0 applications with complex logic often require external dependencies that can only be practically or/and economically satisfied via federated or centralized computing. Per our discussions with industrial practitioners, especially those with actual application scenarios besides token transfers, the off-chain part of their applications may require data collections from distributed IoT sensors deployed in different organizations, machine learning assisted big data analytics, and endorsement or certificates from trusted parties (usually governmental institutes). As a result, it is foreseeable that Web3.0 will not and should not exclude interactions with the federated or centralized platforms.

Therefore, in terms of defining applications for the Web3.0 era (separating them from the so-called Web2.0 applications), it is crucial to draw a line between the part that verifiability is deemed necessary and the part that still depends on the outputs from non-decentralized organizations with proprietary algorithms and/or hardware, and certain certificates. Thus, introducing the abstracted and application-specific concept of critical computing provides a definitive and measurable metric for establishing Web3.0 applications that require non-trivial interactions with these organizations.

2.3 Web3.0 and Its Key Infrastructural Enablers

Taking all our observations and reasoning together, we propose our statement about Web3.0 as: *Web3.0 is an era of computing where the critical computing of applications is verifiable*. We clarify that this statement is not the only way of understanding Web3.0, yet this definition captures the fundamental and defining trait of Web3.0 and has two desirable properties: generic and measurable. Specifically, it is not limited to any specific overarching applications or underlying infrastructures. Crucially, all stakeholders are able to decide whether the execution of an application is verifiable (*i.e.,* measuring the application's eligibility for Web3.0) based on predetermined terms.

Under this definition, we propose three key infrastructural enablers for Web3.0.

(i) Individual smart-contract capable blockchains will continue to play a crucial role, especially after considerable efforts have been made to improve the performance and security of individual blockchains, such as more efficient consensus algorithms [8–10], improving transaction rate by sharding [11–13], enhancing the privacy for smart contracts [14, 15], and reducing their vulnerabilities via program analysis [16–18]. Due to the built-in verifiability of smart contracts, executing the critical computing of Web3.0 applications on-chain as smart contracts is a sensible option.

(ii) A mature Web3.0 era should also include federated or centralized platforms to compensate for functionality that is difficult or infeasible to execute on-chain, *e.g.,* performing computationally intensive tasks or special functionality requiring governmental endorsement. We recognize that two key capabilities are required from those federated or centralized platforms (with minimal disruption of their operational models) to make them compatible with and qualified to be part of Web3.0: (i) any public state they publish should be coupled with verifiable proofs to certify the correctness of the state, where the definition of correctness is application-specific, and (ii) all published state should have the concept of finality. Recent proposals, such as scaling zero-knowledge proofs using distributed clusters [19] or exporting verifiable states from certified websites [20], provide a promising direction in this regard.

(iii) Finally, the critical computing of complex Web3.0 applications often rely on distributed states published by different and isolated systems, including both individual blockchains and federated / central platforms. The reason that multiple state publishers are required in critical computing is not because a single system, after optimization, cannot acquire all required functionality. Instead, it is because these systems are owned by different organizations and therefore it is impossible to merge them. As a result, a *secure interoperability platform* to hyperconnect these isolated state publishers (both decentralized and federated / central ones) is the final building block to enable Web3.0.

Among the three aforementioned categories of key infrastructural enablers, the focus of the remainder of this paper is a design for the third enabler, namely the first interoperability platform for Web3.0, named HyperService. Existing Blockchain interoperability proposals mostly center around atomic token-exchange, which is only a small fraction of the complete scope of Web3.0 applications. HyperService, for the first time, delivers programmability with interoperability, providing a unified and connected computing platform atop heterogeneous blockchains and federated state publishers to power the critical computing for Web3.0 applications.

3 HyperService OVERVIEW

3.1 Architecture

As depicted in Figure 1, architecturally, HyperService is designed around four components. (i) *dApp Clients* are the gateways for dApps to interact with the HyperService platform. When designing HyperService, we intentionally make clients 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

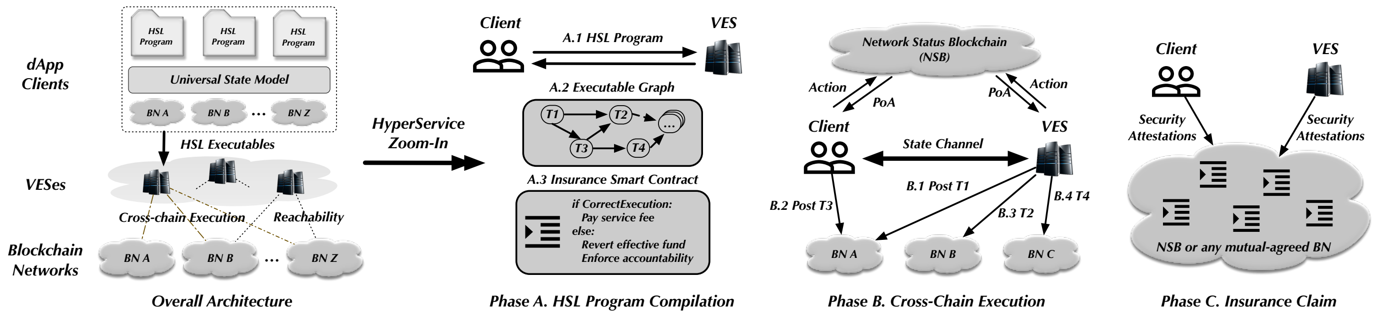

Fig. 1: The architecture of HyperService.

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.

3.2 Universal State Model

A blockchain, together with smart contracts (or dApps) executed on the blockchain, is perceived as a state machine [6]. We desire to preserve the similar abstraction for developers when writing cross-chain dApps. To 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 environment, programming languages), are abstracted as *objects with public state variables and functions*; (ii) developers write dApps by specifying operations over those objects, along with the relative ordering among those operations, as if all the objects were local to a single machine.

Formally, USM is defined as $\mathcal{M} = \{\mathcal{E}, \mathcal{P}, \mathcal{C}\}$ where $\mathcal{E}$ is a set of *entities*, $\mathcal{P}$ is a set of *operations* performed over those entities, and $\mathcal{C}$ is a set of constraints defining the *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. 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, as discussed in § 3.3.

An operation in USM defines a step of computation performed over several entities. Table 1 lists two kinds of operations: a *payment* operation describing the balance updates between two account entities at a certain exchange rate; and an *invocation* operation describing 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 [21], 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.

3.3 HyperService Programming Language

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

3.3.1 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 [22] 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.

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 **16** and **20** in Figure 2). Meanwhile, HSL supports conditional control constructs (*i.e.,* **if** and **else**) to enable intelligent semantics such as only executing (or partially executing) the option terms if the genuine strike price is above a certain target. In addition, looping control constructs are allowed to achieve concise expression of the HSL program. Detailed HSL grammar is given in Grammar 1.

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

```
# Import the source code of contracts written in different languages.
import ("broker.sol", "option.vy", "option.go")
# Entity definition.
# Attributes of a contract entity are implicit from its source code.
account a1 = ChainX::Account(0x7019..., 100, xcoin)
account a2 = ChainY::Account(0x47a1..., 0, ycoin)
account a3 = ChainZ::Account(0x61a2..., 50, zcoin)
account a4 = ChainZ::Account(0x853e..., 50, zcoin)
contract c1 = ChainX::Broker(0xbba7...)
contract c2 = ChainY::Option(0x917f...)
contract c3 = ChainZ::Option(0xefed...)
# Operation definition.
op op1 invocation c1.GetStrikePrice() using a1
if c1.StrikePrice > target :
  op op2 payment 50 xcoin from a1 to a2 with 1 xcoin as 0.5 ycoin
  op op3 invocation c2.CashSettle(10, c1.StrikePrice) using a2
  for acc in [a3, a4] :
    op op4 invocation c3.CashSettle(5, c1.StrikePrice) using acc
else : # the StrikePrice is below the target
  op op5 invocation c3.CashSettle(3, c1.StrikePrice) using a4
# Dependency definition.
op1 before op2, op4, op5; op3 after op2
op1 deadline 10 blocks; op2, op3 deadline default; op4 deadline 20 mins
```

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

3.3.2 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 innovations 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 (§ 4.1). This enables effective correctness checks on the HSL programs (§ 4.3). 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 generates 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 to honor the dependency constraints defined in the HSL program.

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 together with the 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.

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

3.4 Universal Inter-Blockchain Protocol (UIP)

To correctly execute a dApp, all executable transactions in its transaction dependency graph must be finalized on blockchains, and meanwhile their preconditions and deadlines are honored. Some transactions in the graph may be dynamically skipped if their depending conditions are not stratified (*e.g.*, the transaction corresponding to op5 in Figure 2). 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 required.

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 finalized 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 § 5.6, we define the security properties of UIP via an ideal functionality.

3.5 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*) [23]. 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

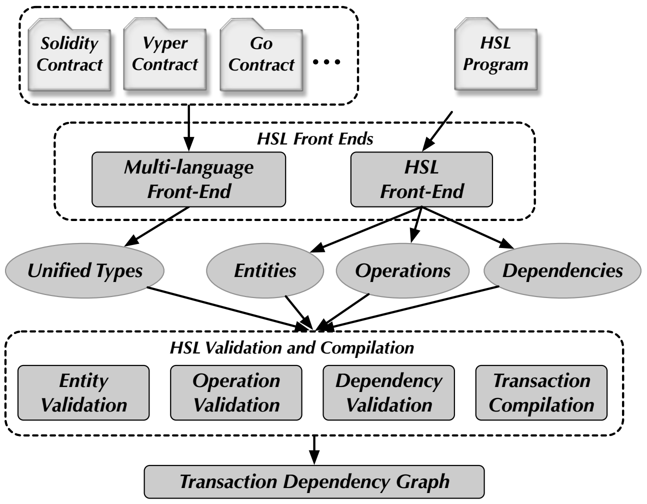

Fig. 3: Workflow of HSL compilation.

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.

4 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 in HSL programs. Afterwards, the compiler performs semantic validations on all entities, operations and dependencies to ensure the security and correctness of the HSL program. Finally, the compiler produces an executable for the HSL program, which is structured in the form of a transaction dependency graph. We next describe the details of each component.

4.1 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* (§ 4.3).

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 blockchains. We therefore omit complex types in HSL.

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

4.2 HSL Language Design

The language constructs provided by HSL are coherent with USM. 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, based on which it performs semantic validation (§ 4.3). 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, as well as a conditional / iteration control construct. 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

```
⟨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_cond⟩ | ⟨op_for⟩ |
                     ⟨op_loop⟩
⟨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⟩
⟨op_block⟩       ::= (⟨entity_def⟩)+ (⟨op_def⟩)+
⟨op_cond⟩        ::= 'if' ⟨cond⟩ ':' ⟨op_block⟩ ('else' ':' ⟨op_block⟩)?
⟨cond⟩           ::= ⟨arg⟩ | ⟨arg⟩ ⟨bop⟩ ⟨arg⟩ | ⟨cond⟩ 'and' ⟨cond⟩ | ⟨cond⟩
                     'or' ⟨cond⟩ | '!' ⟨cond⟩
⟨op_for⟩         ::= 'for' ⟨var⟩ 'in' ⟨collection⟩ ':' ⟨op_block⟩
⟨var⟩            ::= ⟨id⟩
⟨collection⟩     ::= '['((⟨acct⟩)+ | (⟨cont⟩)+)']'
⟨cont⟩           ::= ⟨id⟩
⟨op_loop⟩        ::= 'loop' '(' ⟨loop_cnt⟩ ')' ':' ⟨op_block⟩
⟨loop_cnt⟩       ::= ⟨int⟩

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

⟨*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 (§ 4.3).

The second category of operation in HSL is control constructs. A *conditional* expression (⟨op_cond⟩) specifies the conditional execution of a sequence of operations (⟨op_block⟩), depending on the evaluation of the conditional expression (⟨$cond$⟩). HSL supports both direct evaluation of a boolean variable (⟨arg⟩) or a comparison of variables (⟨arg⟩ ⟨bop⟩ ⟨arg⟩) in a conditional expression. Further, it also supports using the operators and and or to combine multiple conditional expressions and the operator ! to negate a conditional expression. Finally, a *conditional* construct may specify another sequence of operations (*i.e.,* "else block") that will be executed if the conditional expression is evaluated to false. For iteration constructs, HSL supports *for* and *loop*. A *for* construct (⟨op_for⟩) specifies a sequence of operations to be repeated for each element in a collection (⟨$collection$⟩). The elements contained by a collection can be either account entities (⟨$acct$⟩) or contract entities (⟨$cont$⟩). A *loop* construct (⟨op_loop⟩) provides another simple way to specify a sequence of operations to be repeated a fixed number of times (⟨$loop\_cnt$⟩).

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 NSB blocks (⟨*int*⟩ $blocks$) or in absolute time (⟨*int*⟩ ⟨*time_unit*⟩) (*c.f.,* § 4.4).

4.3 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. The same requirement is enforced for the conditional control constructs (*i.e.,* **if** and **else**), *i.e.,* only literals and state variables are eligible in the conditional statement.

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. If the HSL contains a loop statement, a single declared operation could be compiled into multiple operations (*e.g.,* the op4 in Figure 2). In this case, all derived operations will have same dependency constraint as the original one, and meanwhile they do not have mutual dependency.

4.4 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 $\mathcal{T}$:=[from, to, seq, meta], where the pair {from, to} decides the sending and receiving addresses of the on-chain transaction, seq (omitted in Figure 4) represents the sequence number of $\mathcal{T}$ in $\mathcal{G}_T$, and meta stores the structured and customizable metadata for $\mathcal{T}$. Below we explain the fields of meta. First, to achieve financial atomicity, meta must populate a tuple ⟨amt, dst⟩ used in 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 of $\mathcal{G}_T$ aborts whereas $\mathcal{T}$ is committed, the dst account is guaranteed to receive the amount of fund specified in amt. As we shall see in § 5.4, 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 also live on the same blockchain hosting ISC.

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.,* § 5.2.3). Third, a cross-chain payment operation in the HSL

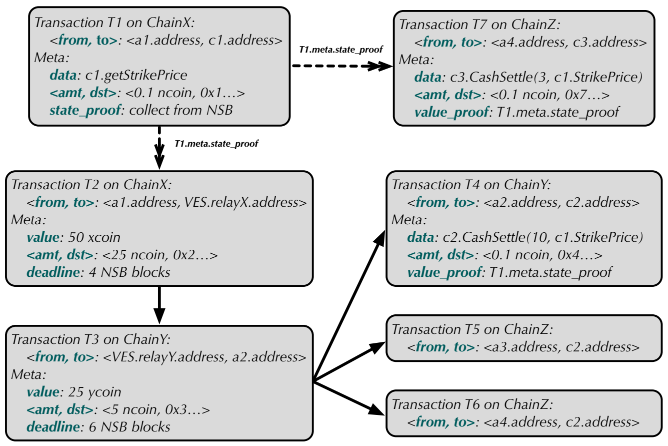
Transaction T1 on ChainX:
<from, to>: <a1.address, c1.address>
Meta:
data: c1.getStrikePrice
<amt, dst>: <0.1 ncoin, 0x1...>
state_proof: collect from NSB
T1.meta.state_proof
Transaction T7 on ChainZ:
<from, to>: <a4.address, c3.address>
Meta:
data: c3.CashSettle(3, c1.StrikePrice)
<amt, dst>: <0.1 ncoin, 0x7...>
value_proof: T1.meta.state_proof
T1.meta.state_proof
Transaction T2 on ChainX:
<from, to>: <a1.address, VES.relayX.address>
Meta:
value: 50 xcoin
<amt, dst>: <25 ncoin, 0x2...>
deadline: 4 NSB blocks
Transaction T4 on ChainY:
<from, to>: <a2.address, c2.address>
Meta:
data: c2.CashSettle(10, c1.StrikePrice)
<amt, dst>: <0.1 ncoin, 0x4...>
value_proof: T1.meta.state_proof
Transaction T3 on ChainY:
<from, to>: <VES.relayY.address, a2.address>
Meta:
value: 25 ycoin
<amt, dst>: <5 ncoin, 0x3...>
deadline: 6 NSB blocks
Transaction T5 on ChainZ:
<from, to>: <a3.address, c2.address>
Transaction T6 on ChainZ:
<from, to>: <a4.address, c2.address>

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

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 § 5.2.2. Transaction deadlines are indeed enforced by the ISC using the number of NSB blocks. 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.

The execution of transactions connected by dotted lines in the $\mathcal{G}_T$ is dynamically decided based on the resulting state of upstream transactions. For instance, in Figure 4 whether *T2* (and all its downstream transactions) or *T7* will be executed is decided by the state after *T1* is finalized.

In summary, the executable produced by the HSL compiler 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.

5 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.

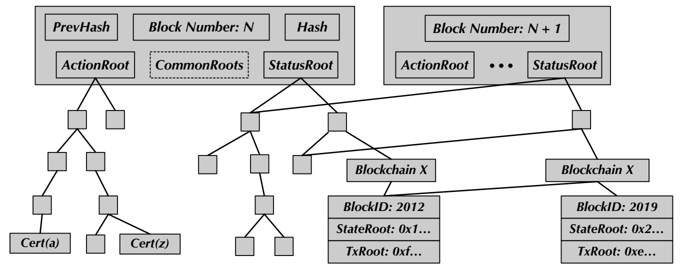

Fig. 5: The architecture of NSB blocks.

5.1 Protocol Preliminaries

5.1.1 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.

5.1.2 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 § 5.1.1, 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.

5.1.3 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.

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.

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*, ensuring that the NSB is horizontally scalable as HyperService incorporates new blockchains. $\mathsf{Prot}_{\mathsf{NSB}}$, discussed in § 5.5, is the protocol that specifies the detailed construction of both roots and guarantees their correctness.

5.2 Execution Protocol by VESes

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

5.2.1 Post Compilation and Session Setup

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

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

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

where $\mathcal{G}_S$ is the set of all committable subsets in $\mathcal{G}_T$, where a subset $s \subseteq \mathcal{G}_T$ is *committable* if, whenever $\mathcal{T} \in s$, all preconditions of $\mathcal{T}$ are also in s. All vertexes in $\mathcal{G}_T$, including those connected by dotted edges, should be included. For clarity of notation, throughout the paper, when saying $\mathcal{T}.\mathsf{from} = \mathsf{Prot}_{\mathsf{VES}}$ or $\mathcal{T}$ is originated from $\mathsf{Prot}_{\mathsf{VES}}$, we mean that $\mathcal{T}$ is sent and signed by an account owned by $\mathsf{Prot}_{\mathsf{VES}}$. Likewise, $\mathcal{T}.\mathsf{from} = \mathsf{Prot}_{\mathsf{CLI}}$ indicates that $\mathcal{T}$ is sent from an account entity defined in the HSL program. $\mathsf{Prot}_{\mathsf{ISC}}$ refunds any remaining funds after the contract is terminated.

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

5.2.2 Protocol Exchange for Transaction Handling

In $\mathsf{Prot}_{\mathsf{VES}}$, SInitedTrans and OpenTrans are two handlers processing *northbound* transactions which originates from $\mathsf{Prot}_{\mathsf{VES}}$. The SInitedTrans handling for $\mathcal{T}$ is invoked when all its preconditions are finalized, which is detected by the watching service of $\mathsf{Prot}_{\mathsf{VES}}$ (*c.f.*, § 5.2.3). The SInitedTrans computes $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ to prove $\mathcal{T}$ is in the inited state , and then passes it to the corresponding handler of $\mathsf{Prot}_{\mathsf{CLI}}$ for subsequent processing. Meanwhile, SInitedTrans stakes $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ on $\mathsf{Prot}_{\mathsf{NSB}}$, and later it retrieves a Merkle proof $\mathsf{Merk}^{i}_{\mathcal{T}}$ from the NSB to prove that $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ has been sent. $\mathsf{Merk}^{\mathsf{id}}_{\mathcal{T}}$ essentially is a hash chain linking $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ back to an ActionRoot on a committed block of the NSB. The proof retrieval is a non-blocking operation triggered by the consensus update on the NSB.

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

Southbound transactions originating from $\mathsf{Prot}_{\mathsf{CLI}}$ are processed by $\mathsf{Prot}_{\mathsf{VES}}$ 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), $\mathsf{Prot}_{\mathsf{VES}}$ needs to verify that the state used by $\widetilde{T}$ is consistent with the state enclosed in the finalization proofs of its upstream transactions.

5.2.3 Proactive Watching Services

Cross-chain execution makes forward progress when session-relevant blockchains and the NSB make progress on transactions. As the driver of execution, $\mathsf{Prot}_{\mathsf{VES}}$ internally creates two watching services to *proactively* read the status of those blockchains.

In the watching daemon to one blockchain, $\mathsf{Prot}_{\mathsf{VES}}$ mainly reads the public ledger of $\mathsf{Prot}_{\mathsf{BC}}$ to monitor the status of transactions that have been posted for on-chain execution. If $\mathsf{Prot}_{\mathsf{VES}}$ notices that an on-chain transaction $\widetilde{T}$ is recently finalized, it requests the closing process for $\mathcal{T}$ by sending $\mathsf{Prot}_{\mathsf{CLI}}$ a timestamped certificate C_{closed}. The pair of handlers, CloseTrans and ClosedTrans, are used by both $\mathsf{Prot}_{\mathsf{VES}}$ and $\mathsf{Prot}_{\mathsf{CLI}}$ 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, $\mathsf{Prot}_{\mathsf{VES}}$ needs to retrieve a Merkle Proof from $\mathsf{Prot}_{\mathsf{BC}}$ to prove the finalization of $\widetilde{T}$. This proof, denoted by $\mathsf{Merk}^{\mathsf{c1}}_{\mathcal{T}}$, 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

```
Init: Data := $\emptyset$
Daemon PostCompiliation():
    generate the session ID sid $\leftarrow \{0,1\}^\lambda$
    call [cid, contract] := $\text{Prot}_\text{ISC}$.CreateContract($\mathcal{G}_\mathcal{T}$)
    send Cert([sid, $\mathcal{G}_\mathcal{T}$, contract]; $\text{Sig}^\mathcal{V}_\text{sid}$) to $\text{Prot}_\text{CLI}$ for approval
    halt until Cert([sid, $\mathcal{G}_\mathcal{T}$, contract]; $\text{Sig}^\mathcal{V}_\text{sid}$, $\text{Sig}^\mathcal{D}_\text{sid}$) is received
    package contract as a valid transaction $\widehat{contract}$
    call $\text{Prot}_\text{NSB}$.Exec($\widehat{contract}$) to deploy the $\widehat{contract}$
    halt until $\widehat{contract}$ is initialized on $\text{Prot}_\text{NSB}$
    call $\text{Prot}_\text{ISC}$.StakeFund to stake the required funds in $\text{Prot}_\text{ISC}$
    halt until $\mathcal{D}$ has staked its required funds in $\text{Prot}_\text{ISC}$
    initialize Data[sid] := {$\mathcal{G}_\mathcal{T}$, cid, $S_\text{Cert}=\emptyset$, $S_\text{Merk}=\emptyset$}
Daemon Watching(sid, {$\text{Prot}_\text{BC}$, ...}) private:
    ($\mathcal{G}_\mathcal{T}$, _, $S_\text{Cert}$, $S_\text{Merk}$) := Data[sid]; abort if not found
    for each $\mathcal{T} \in \mathcal{G}_\mathcal{T}$ :
        continue if $\mathcal{T}$.state is not opened
        identify $\mathcal{T}$'s on-chain counterpart $\widetilde{\mathcal{T}}$
        continue if $\text{Prot}_\text{BC}$.Status($\widetilde{\mathcal{T}}$) is not committed
        get $\text{ts}_\text{closed}$ := $\text{Prot}_\text{NSB}$.BlockHeight()
        compute $C^\mathcal{T}_\text{closed}$ := Cert([$\widetilde{\mathcal{T}}$, closed, sid, $\mathcal{T}$, $\text{ts}_\text{closed}$], $\text{Sig}^\mathcal{V}_\text{sid}$)
        call $\text{Prot}_\text{CLI}$.CloseTrans($C^\mathcal{T}_\text{closed}$) to negotiate the closed attestation
        call $\text{Prot}_\text{BC}$.MerkleProof($\widetilde{\mathcal{T}}$) to obtain a finalization proof for $\widetilde{\mathcal{T}}$
        denote the finalization proof as $\text{Merk}^{c1}_\mathcal{T}$ (Figure 7)
        update $S_\text{Cert}$.Add($C^\mathcal{T}_\text{closed}$) and $S_\text{Merk}$.Add($\text{Merk}^{c1}_\mathcal{T}$)
Daemon Watching(sid, $\text{Prot}_\text{NSB}$) private:
    ($\mathcal{G}_\mathcal{T}$, _, $S_\text{Cert}$, $S_\text{Merk}$) := Data[sid]; abort if not found
    watch four types of attestations {$\text{Cert}^\text{id}$, $\text{Cert}^o$, $\text{Cert}^\text{od}$, $\text{Cert}^c$}
    process fresh attestations via corresponding handlers (see below)
    # Retrieve alternative attestations if necessary.
    for each $\mathcal{T} \in \mathcal{G}_\mathcal{T}$ :
        if $\mathcal{T}$.state = opened and $\text{Merk}^{c1}_\mathcal{T} \in S_\text{Merk}$ :
            retrieve the roots [R, ...] of the proof $\text{Merk}^{c1}_\mathcal{T}$
            call $\text{Prot}_\text{NSB}$.MerkleProof([R, ...]) to obtain a status proof $\text{Merk}^{c2}_\mathcal{T}$
            continue if $\text{Merk}^{c2}_\mathcal{T}$ is not available yet on $\text{Prot}_\text{NSB}$
            compute the complete proof $\text{Merk}^{c}_\mathcal{T}$ := [$\text{Merk}^{c1}_\mathcal{T}$, $\text{Merk}^{c2}_\mathcal{T}$]
            update $\mathcal{T}$.state := closed and $S_\text{Merk}$.Add($\text{Merk}^{c}_\mathcal{T}$)
    compute eligible transaction set $\mathcal{S}$ using the current state of $\mathcal{G}_\mathcal{T}$
    for each $\mathcal{T} \in \mathcal{S}$:
        continue if $\mathcal{T}$.state is not unknown
        if $\mathcal{T}$.from = $\text{Prot}_\text{CLI}$:
            compute $\text{Cert}^i_\mathcal{T}$ := Cert([$\mathcal{T}$, init, sid]; $\text{Sig}^\mathcal{V}_\text{sid}$)
            call $\text{Prot}_\text{CLI}$.InitTrans($\text{Cert}^i_\mathcal{T}$) to request initialization
            call $\text{Prot}_\text{NSB}$.AddAction($\text{Cert}^i_\mathcal{T}$) to prove $\text{Cert}^i_\mathcal{T}$ is sent
            update $S_\text{Cert}$.Add($\text{Cert}^i_\mathcal{T}$) and $\mathcal{T}$.state := init
            non-blocking wait until $\text{Prot}_\text{NSB}$.MerkleProof($\text{Cert}^i_\mathcal{T}$) rt. $\text{Merk}^i_\mathcal{T}$
            update $S_\text{Merk}$.Add($\text{Merk}^i_\mathcal{T}$)
        else: call self.SInitedTrans(sid, $\mathcal{T}$)
Upon Receive SInitedTrans(sid, $\mathcal{T}$) private:                                Northbound
    ($\mathcal{G}_\mathcal{T}$, _, $S_\text{Cert}$, $S_\text{Merk}$) := Data[sid]; abort if not found
    compute and sign the on-chain counterpart $\widetilde{\mathcal{T}}$ for $\mathcal{T}$
    compute $\text{Cert}^\text{id}_\mathcal{T}$ := Cert([$\widetilde{\mathcal{T}}$, inited, sid, $\mathcal{T}$]; $\text{Sig}^\mathcal{V}_\text{sid}$)
    call $\text{Prot}_\text{CLI}$.InitedTrans($\text{Cert}^\text{id}_\mathcal{T}$) to request opening of initialized $\mathcal{T}$
    call $\text{Prot}_\text{NSB}$.AddAction($\text{Cert}^\text{id}_\mathcal{T}$) to prove $\text{Cert}^\text{id}_\mathcal{T}$ is sent
    update $S_\text{Cert}$.Add($\text{Cert}^\text{id}_\mathcal{T}$) and $\mathcal{T}$.state := inited
    non-blocking wait until $\text{Prot}_\text{NSB}$.MerkleProof($\text{Cert}^\text{id}_\mathcal{T}$) returns $\text{Merk}^\text{id}_\mathcal{T}$
    update $S_\text{Merk}$.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([$\widetilde{\mathcal{T}}$, inited, sid, $\mathcal{T}$]; $\text{Sig}^\mathcal{D}_\text{sid}$)
    (_, _, $S_\text{Cert}$, $S_\text{Merk}$) := 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 $S_\text{Cert}$
    assert $\widetilde{\mathcal{T}}$ is correctly associated with the wrapper $\mathcal{T}$
    get $\text{ts}_\text{open}$ := $\text{Prot}_\text{NSB}$.BlockHeight()
    compute $\text{Cert}^o_\mathcal{T}$ := Cert([$\widetilde{\mathcal{T}}$, open, sid, $\mathcal{T}$, $\text{ts}_\text{open}$]; $\text{Sig}^\mathcal{V}_\text{sid}$)
    call $\text{Prot}_\text{CLI}$.OpenTrans($\text{Cert}^o_\mathcal{T}$) to request opening for $\mathcal{T}$
    call $\text{Prot}_\text{NSB}$.AddAction($\text{Cert}^o_\mathcal{T}$) to prove $\text{Cert}^o_\mathcal{T}$ is sent
    update $S_\text{Cert}$.Add($\text{Cert}^o_\mathcal{T}$) and $\mathcal{T}$.state := open
    non-blocking wait until $\text{Prot}_\text{NSB}$.MerkleProof($\text{Cert}^o_\mathcal{T}$) returns $\text{Merk}^o_\mathcal{T}$
    update $S_\text{Merk}$.Add($\text{Merk}^o_\mathcal{T}$)
Upon Receive OpenTrans($\text{Cert}^o_\mathcal{T}$) public:                                Northbound
    assert $\text{Cert}^o_\mathcal{T}$ has valid form of Cert([$\widetilde{\mathcal{T}}$, open, sid, $\mathcal{T}$, $\text{ts}_\text{open}$]; $\text{Sig}^\mathcal{D}_\text{sid}$)
    (_, _, $S_\text{Cert}$, $S_\text{Merk}$) := Data[sid]; abort if not found
    abort if the $\text{Cert}^\text{id}_\mathcal{T}$ corresponding to $\text{Cert}^o_\mathcal{T}$ is not in $S_\text{Cert}$
    assert $\text{ts}_\text{open}$ is within a bounded range with $\text{Prot}_\text{NSB}$.BlockHeight()
    compute $\text{Cert}^\text{od}_\mathcal{T}$ := Cert([$\widetilde{\mathcal{T}}$, open, sid, $\mathcal{T}$, $\text{ts}_\text{open}$]; $\text{Sig}^\mathcal{D}_\text{sid}$, $\text{Sig}^\mathcal{V}_\text{sid}$)
    call $\text{Prot}_\text{BC}$.Exec($\widetilde{\mathcal{T}}$) to trigger on-chain execution
    call $\text{Prot}_\text{CLI}$.OpenedTrans($\text{Cert}^\text{od}_\mathcal{T}$) to acknowledge request
    call $\text{Prot}_\text{NSB}$.AddAction($\text{Cert}^\text{od}_\mathcal{T}$) to prove $\text{Cert}^\text{od}_\mathcal{T}$ is sent
    update $S_\text{Cert}$.Add($\text{Cert}^\text{od}_\mathcal{T}$) and $\mathcal{T}$.state := opened
    non-blocking wait until $\text{Prot}_\text{NSB}$.MerkleProof($\text{Cert}^\text{od}_\mathcal{T}$) returns $\text{Merk}^\text{od}_\mathcal{T}$
    update $S_\text{Merk}$.Add($\text{Merk}^\text{od}_\mathcal{T}$)
Upon Receive OpenedTrans($\text{Cert}^\text{od}_\mathcal{T}$) public:                        Southbound
    ast. $\text{Cert}^\text{od}_\mathcal{T}$ has valid form of Cert([$\widetilde{\mathcal{T}}$, open, sid, $\mathcal{T}$, $\text{ts}_\text{open}$]; $\text{Sig}^\mathcal{V}_\text{sid}$, $\text{Sig}^\mathcal{D}_\text{sid}$)
    (_, _, $S_\text{Cert}$, _) := Data[sid]; abort if not found
    abort if the $\text{Cert}^o_\mathcal{T}$ corresponding to $\text{Cert}^\text{od}_\mathcal{T}$ is not in $S_\text{Cert}$
    update $S_\text{Cert}$.Add($\text{Cert}^\text{od}_\mathcal{T}$) 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([$\widetilde{\mathcal{T}}$, closed, sid, $\mathcal{T}$, $\text{ts}_\text{closed}$], $\text{Sig}^\mathcal{D}_\text{sid}$)
    assert $\widetilde{\mathcal{T}}$ is finalized on its destination blockchain and obtain $\text{Merk}^{c1}_\mathcal{T}$
    assert $\text{ts}_\text{closed}$ is within a bounded margin with $\text{Prot}_\text{NSB}$.BlockHeight()
    (_, _, $S_\text{Cert}$, $S_\text{Merk}$) := Data[sid]; abort if not found
    compute $\text{Cert}^c_\mathcal{T}$ := Cert([$\widetilde{\mathcal{T}}$, closed, sid, $\mathcal{T}$, $\text{ts}_\text{closed}$], $\text{Sig}^\mathcal{D}_\text{sid}$, $\text{Sig}^\mathcal{V}_\text{sid}$)
    call $\text{Prot}_\text{CLI}$.ClosedTrans($\text{Cert}^c_\mathcal{T}$) to acknowledged request
    update $S_\text{Cert}$.Add($\text{Cert}^c_\mathcal{T}$), $S_\text{Merk}$.Add($\text{Merk}^{c1}_\mathcal{T}$) and $\mathcal{T}$.state := closed
Upon Receive ClosedTrans($\text{Cert}^c_\mathcal{T}$) public:                             Bidirectional
    ast. $\text{Cert}^c_\mathcal{T}$ has valid form of Cert([$\widetilde{\mathcal{T}}$, closed, sid, $\mathcal{T}$, $\text{ts}_\text{closed}$], $\text{Sig}^\mathcal{V}_\text{sid}$, $\text{Sig}^\mathcal{D}_\text{sid}$)
    (_, _, $S_\text{Cert}$, _) := Data[sid]; abort if not found
    abort if Cert([$\widetilde{\mathcal{T}}$, closed, sid, $\mathcal{T}$, $\text{ts}_\text{closed}$], $\text{Sig}^\mathcal{V}_\text{sid}$) is not in $S_\text{Cert}$
    update $S_\text{Cert}$.Add($\text{Cert}^c_\mathcal{T}$) and $\mathcal{T}$.state := closed
Daemon Redeem(sid) private:
    # Invoke the insurance contract periodically
    ($\mathcal{G}_\mathcal{T}$, cid, $S_\text{Cert}$, $S_\text{Merk}$) := Data[sid]; abort if not found
    for each unclaimed $\mathcal{T} \in \mathcal{G}_\mathcal{T}$:
        get the $\text{Cert}_\mathcal{T}$ from $S_\text{Cert} \bigcup S_\text{Merk}$ with the most advanced state
        call $\text{Prot}_\text{ISC}$.InsuranceClaim(cid, $\text{Cert}_\mathcal{T}$) to claim insurance
```

Fig. 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.

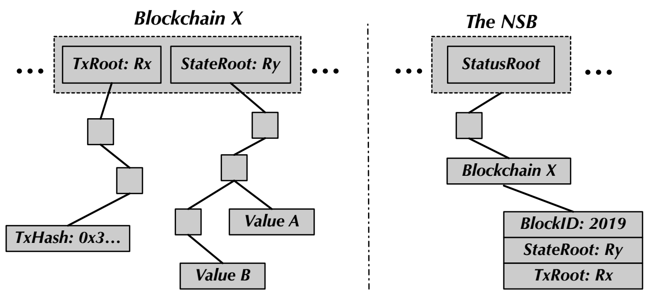

Fig. 7: The complete on-chain proof (denoted by $\text{Merk}^c_\mathcal{T}$) to prove that the state of a transaction is eligible to be promoted as closed. The left-side part is the finalization proof (denoted by $\text{Merk}^{c1}_\mathcal{T}$) for the transaction collected from its destination blockchain; the right-side part is the blockchain status proof (denoted by $\text{Merk}^{c2}_\mathcal{T}$) collected from the NSB.

closed, as shown in Figure 7; (ii) if the resulting state of $\widetilde{\mathcal{T}}$ is used by its downstream transactions, $\text{Merk}^{c1}_\mathcal{T}$ is necessary to ensure that those downstream transactions indeed use the genuine state.

In the watching service to Prot_NSB, Prot_VES performs following tasks. First, as described in § 5.1.2, NSB is treated as a fallback communication medium for the off-chain channel. Thus, Prot_VES searches the sorted ActionMT to look for any session-relevant certificates that have not been received via the off-chain channel. Second, for each opened $\mathcal{T}$ whose closed attestation is still missing after Prot_VES has sent C_closed (indicating slow or no reaction from Prot_CLI), Prot_VES tries to retrieve the second part of $\text{Merk}^c_\mathcal{T}$ from Prot_NSB. The second proof, denoted as $\text{Merk}^{c2}_\mathcal{T}$, is to prove that the Merkle roots referred in $\text{Merk}^{c1}_\mathcal{T}$ are correctly linked to a StatusRoot on a finalized NSB block (see Figure 7). Once $\text{Merk}^c_\mathcal{T}$ is fully constructed, the state of $\mathcal{T}$ is promoted as closed. Finally, Prot_VES may find a new set of transactions that are eligible to be executed if their preconditions are finalized due to any recently-closed transactions. If so, Prot_VES

```
Init: Data := $\emptyset$
Upon Receive CreateContract($\mathcal{G}_T$):
  generate the arbitration cod, denoted by contract, as follows
  initialize three maps $\mathsf{T}_{state}$, $\mathsf{A}_{revs}$ and $\mathsf{F}_{stake}$
  for each $\mathcal{T} \in \mathcal{G}_T$ :
    compute an internal identifier for $\mathcal{T}$ as tid := $H(\mathcal{T})$
    initialize $\mathsf{T}_{state}$[tid] := [unknown, $\mathcal{T}$, $ts_{open}$=0, $ts_{closed}$=0, $st_{proof}$]
    retrieve tid's fund-reversion account, denoted as dst
    initialize $\mathsf{A}_{revs}$[tid] := [amt=0, dst]
  compute an identifier for contract as cid := $H(\overrightarrow{0}$, contract)
  initialize Data[cid] := [$\mathcal{G}_T$, $\mathsf{T}_{state}$, $\mathsf{A}_{revs}$, $\mathsf{F}_{stake}$]
  send [cid, contract] to the requester for acknowledgment
Upon Receive StakeFund(cid):
  (_, _, _, $\mathsf{F}_{stake}$) := Data[cid]; abort if not found
  update $\mathsf{F}_{stake}$[msg.sender] := $\mathsf{F}_{stake}$[msg.sender] + msg.value
Upon Receive InsuranceClaim(cid, Atte):
  ($\mathcal{G}_T$, _, $\mathsf{T}_{state}$, _, _) := Data[cid]; abort if not found
  compute tid := $H$(Atte.$\mathcal{T}$); T := $\mathsf{T}_{state}$[tid] abort if not found
  abort if T.state is more advanced the state enclosed by Cert
  abort if T is unreachable based on current state of $\mathcal{G}_T$
  if Atte is a certificate signed by both parties :
    assert SigVerify(Atte) is true
    if Atte is $\mathsf{Cert}^{od}_{\mathcal{T}}$ : update T.state := opened; T.$ts_{open}$ := Atte.$ts_{open}$
    else : update T.state := closed; T.$ts_{closed}$ := Atte.$ts_{closed}$
  else :  # Atte is in form of a Merkle proof
    assert MerkleVerify(Atte) is true
    if Atte is a $\mathsf{Merk}^{i}_{\mathcal{T}}$ or $\mathsf{Merk}^{id}_{\mathcal{T}}$ or $\mathsf{Merk}^{o}_{\mathcal{T}}$ :
      retrieve the certificate $\mathsf{Cert}^{i}_{\mathcal{T}}$ or $\mathsf{Cert}^{id}_{\mathcal{T}}$ or $\mathsf{Cert}^{o}_{\mathcal{T}}$ from Atte
      assert the $\widetilde{T}$ enclosed in $\mathsf{Cert}^{id}_{\mathcal{T}}$ or $\mathsf{Cert}^{o}_{\mathcal{T}}$ is genuine
      assert the $ts_{open}$ enclosed in $\mathsf{Cert}^{o}_{\mathcal{T}}$ is genuine
      update T.state := Atte.state
    elif Atte is $\mathsf{Merk}^{od}_{\mathcal{T}}$ :
      retrieve the certificate $\mathsf{Cert}^{od}_{\mathcal{T}}$ from Atte
      update T.state := opened and T.$ts_{open}$ := $\mathsf{Cert}^{od}_{\mathcal{T}}$.$ts_{open}$
    elif Atte is $\mathsf{Merk}^{c}_{\mathcal{T}}$ :
      update T.$st_{proof}$ based on $\mathsf{Merk}^{c1}_{\mathcal{T}}$ if necessary
      update T.$ts_{closed}$ as the height of the block attaching $\mathsf{Merk}^{c2}_{\mathcal{T}}$
      update T.state := closed
Upon Timeout SettleContract(cid):                              Internal Daemon
  ($\mathcal{G}_T$, $\mathsf{T}_{state}$, $\mathsf{A}_{revs}$, $\mathsf{F}_{stake}$) := Data[cid]; abort if not found
  for (tid, T) $\in$ $\mathsf{T}_{state}$ :
    continue if T.state is not closed
    update $\mathsf{A}_{revs}$[tid].amt := T.$\mathcal{T}$.meta.amt
    if DeadlineVerify(T) = true : update T.state := correct
  compute $\mathcal{S}$ := DirtyTrans($\mathcal{G}_T$, $\mathsf{T}_{state}$) # non-empty if execution fails.
  execute fund reversion for non-zero entries in $\mathsf{A}_{revs}$ if $\mathcal{S}$ is not empty
  initialize a map resp to record which party to blame
  for each (tid, T) $\in \mathcal{S}$ :
    if T.state = closed | open | opened : resp[tid] := T.$\mathcal{T}$.from
    elif T.state = inited : resp[tid] := T.$\mathcal{T}$.to
    elif T.state = init : resp[tid] := $\mathcal{D}$
    else : resp[tid] := $\mathcal{V}$
  return any remaining funds in $\mathsf{F}_{stake}$ to corresponding senders
  call Data.erase[cid] to stay silent afterwards
```

Fig. 8: Prot_{ISC}: the protocol realization of the ISC arbitrator.

processes them by either requesting initialization from Prot_{CLI} or calling SInitedTrans internally, depending on the originators of those transactions.

5.2.4 Prot_{ISC} *Invocation*

All internally stored certificates and *complete* Merkle proofs are acceptable by Prot_{ISC} to execute contract terms. However, for any $\mathcal{T}$, Prot_{VES} should invoke Prot_{ISC} only using the attestation with the most advanced state, since lower-ranked attestations for $\mathcal{T}$ are effectively ignored by Prot_{ISC} (*c.f.*, § 5.4).

5.3 Execution Protocol by dApp Clients

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

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

5.4 Protocol Realization of the ISC

Figure 8 specifies the protocol realization of the ISC. The CreateContract handler is the entry point of requesting insurance contract creation using Prot_{ISC}. It generates the arbitration code, denoted as *contract*, based on the given dApp executable $\mathcal{G}_T$. The *contract* internally uses T_{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 st_{proof} and A_{revs}. When the Prot_{ISC} times out, it executes the contract terms based on its internal state, after which its funds are depleted and the contract never runs again. Below we explain several technical subtleties.

5.4.1 *Insurance Claim*

The InsuranceClaim handler processes security attestations from Prot_{VES} and Prot_{CLI}. When receiving an attestation for a transaction $\mathcal{T}$, it first verifies that $\mathcal{T}$ is *reachable* based on the current state of the $\mathcal{G}_T$, *i.e.*, all its upstream transactions are finalized. Thus, if a branch in $\mathcal{G}_T$ is dynamically pruned because of unsatisfied conditions (*e.g.*, *T7* in Figure 4), all transactions in the branch are effectively skipped. InsuranceClaim only accept dual-signed certificates (*i.e.*, Cert^{od} and Cert^{c}) or complete Merkle proofs. 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 $\mathsf{Merk}^{i}_{\mathcal{T}}$, $\mathsf{Merk}^{id}_{\mathcal{T}}$ or $\mathsf{Merk}^{o}_{\mathcal{T}}$, Prot_{ISC} retrieves the single-party signed certificate $\mathsf{Cert}^{i}_{\mathcal{T}}$, $\mathsf{Cert}^{id}_{\mathcal{T}}$ or $\mathsf{Cert}^{o}_{\mathcal{T}}$ 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.*, $\mathsf{Cert}^{i}_{\mathcal{T}}$ is signed by Prot_{VES}, $\mathsf{Cert}^{id}_{\mathcal{T}}$ is signed by $\mathcal{T}$'s originator and $\mathsf{Cert}^{o}_{\mathcal{T}}$ is signed by the destination of $\mathcal{T}$. (ii) The enclosed on-chain transaction $\widetilde{T}$ in $\mathsf{Cert}^{id}_{\mathcal{T}}$ and $\mathsf{Cert}^{o}_{\mathcal{T}}$ is correctly associated with $\mathcal{T}$. The checking logic is the same as the on used by Prot_{VES}, which has been explained in § 5.2.2. (iii) The enclosed ts_{open} in $\mathsf{Cert}^{o}_{\mathcal{T}}$ is genuine, where the genuineness is defined as a bounded difference between ts_{open} and the height of the NSB block that attaches $\mathsf{Merk}^{o}_{\mathcal{T}}$.

5.4.2 *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

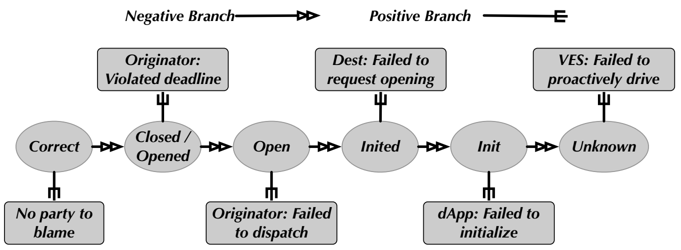
Negative Branch
Positive Branch
Originator: Violated deadline
Dest: Failed to request opening
VES: Failed to proactively drive
Correct
Closed / Opened
Open
Inited
Init
Unknown
No party to blame
Originator: Failed to dispatch
dApp: Failed to initialize

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

if $\mathcal{G}_T$ has no dirty transactions. Otherwise, $\mathsf{Prot}_{\mathsf{ISC}}$ 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 $\mathsf{Prot}_{\mathsf{VES}}$ and $\mathsf{Prot}_{\mathsf{CLI}}$. 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 $\mathsf{Cert}^{\mathsf{id}}_{\mathcal{T}}$ has been provably sent. If a transaction $\mathcal{T}$'s state is init (only transactions originated from dApp $\mathcal{D}$ can have init status), then $\mathcal{D}$ (the originator) is the party to blame for not reacting on the $\mathsf{Cert}^{i}_{\mathcal{T}}$ sent by $\mathcal{V}$. Finally, if transaction $\mathcal{T}$'s state is unknown, then $\mathcal{V}$ is held accountable for not proactively driving the initialization of $\mathcal{T}$, no matter which party originates $\mathcal{T}$.

5.5 Specification of $\mathsf{Prot}_{\mathsf{NSB}}$ and $\mathsf{Prot}_{\mathsf{BC}}$

$\mathsf{Prot}_{\mathsf{BC}}$ specifies the protocol realization of a general-purpose blockchain where a set of consensus nodes run a secure protocol to agree upon the public global state. In this paper, we regard $\mathsf{Prot}_{\mathsf{BC}}$ as a conceptual party trusted for correctness and availability, *i.e.*, $\mathsf{Prot}_{\mathsf{BC}}$ guarantees to correctly perform any predefined computation (*e.g.*, Turing-complete smart contract programs) and is always available to handle user requests despite unbounded response latency. $\mathsf{Prot}_{\mathsf{NSB}}$ specifies the protocol realization of the NSB. $\mathsf{Prot}_{\mathsf{NSB}}$ is an extended version of $\mathsf{Prot}_{\mathsf{BC}}$ with additional capabilities. Due to space constraint, we defer the detailed protocol description of $\mathsf{Prot}_{\mathsf{BC}}$ and $\mathsf{Prot}_{\mathsf{NSB}}$ to our technical report [24].

5.6 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 $\mathcal{F}_{\mathsf{UIP}}$. The ideal functionality articulates the correctness and security properties that UIP wishes to attain by assuming a trusted entity. Then we prove that $\mathsf{Prot}_{\mathsf{UIP}}$, our the decentralized real-world protocol containing the aforementioned preliminary protocols, securely realizes $\mathcal{F}_{\mathsf{UIP}}$ using the UC framework [25], *i.e.*, $\mathsf{Prot}_{\mathsf{UIP}}$ achieves the same functionality and security properties as $\mathcal{F}_{\mathsf{UIP}}$ without assuming any trusted authorities. The detailed proof is available in [1].

5.7 Hyperconnecting Federated / Centralized Platforms

As discussed in § 2.3, Web3.0 should also include federated or centralized platforms that are able to publish verifiable state and the published state has the concept of finality. Thanks to the infrastructural abstraction used in HyperService (although the abstraction is originally designed to interoperate heterogeneous blockchains), extending HyperService to include these non-decentralized state publishers is promising. First, in USM, a state publisher can be represented as a new abstract entity *publisher* with publicly callable interfaces whose internal implementation logic is possibly proprietary (and therefore could be confidential). Because of such opacity, a *publisher* needs to provide an auxiliary proof for any output state of an interface so that the state is qualified to be used in HSL programs (recall that the HSL compiler performs verifiability check when compiling HSL programs (§ 4.3)). Second, if any inputs are taken by the interface to compute output state, the inputs should be also attached to the output state proof. This is because the requests to execute a contract interface are currently driven by blockchain transactions so that all inputs to the interface are persistent on blockchains. When this transaction-centric model is not applicable to some state publishers, the linkability between inputs and outputs is broken. Thus, attaching inputs to output state proofs reestablishes the linkability, allowing UIP to verify that the downstream operations (could be either blockchain transactions or requests sent to state publishers) use the correct state resulting from upstream operations.

6 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 [26] includes (i) ∼2,400 lines of Java code and ∼3,300 lines of ANTLR [27] grammar code for building the HSL programming framework, (ii) ∼41,000 lines of code, mainly in Go and Python, for implementing the UIP protocol; and ∼15,000 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.

6.1 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 [28] 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 a HSL compiler that takes HSL programs and contracts written in Solidity, Vyper, and Go as input, and produces TDGs. We implement the multi-lang frontend and the HSL frontend using ANTLR [27], 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.

6.2 Application Implementation

Besides the platform implementation, we also 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 instruc-

	Financial Derivatives		CryptoAsset Movement		Federated Computing	
	Mean	%	Mean	%	Mean	%
HSL Compilation	1.2317	~14	0.2995	~5	1.1417	~22
Session Creation	5.6910	~63	3.6640	~ 61	2.0320	~39
Action/Status Staking	1.0295	~11	1.0178	~17	1.0163	~19
Proof Retrieval	1.0214	~11	1.0167	~17	1.0192	~19
Total	8.9736		5.9980		5.2092	

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

tions. Currently, oracles are either built atop trusted third-party providers (*e.g.*, Oraclize [29]), or using trusted hardware enclaves [22]. 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 § 3.3).

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 [30], 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.

6.3 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.

6.3.1 End-to-End Latency

We evaluated all three applications mentioned in § 6.2 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.

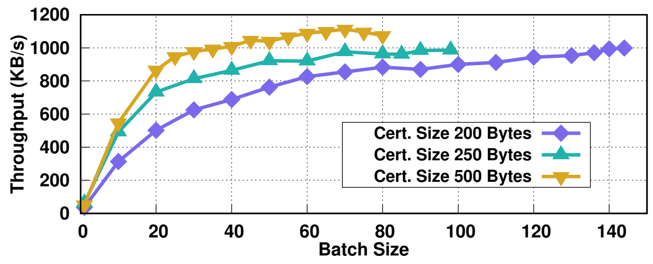

Fig. 10: The throughput of the NSB, measured as the total size of committed certificates on the NSB per second.

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.

6.3.2 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 1000 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 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 a certificate size, further enlarging the batch size cannot boost 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 (§ 5.1.3) 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.

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 [2, 5, 31], atomic cross-chain swaps [3, 7], and cryptocurrency-backed assets [4]. However, programmability, *i.e.,* smart contracting across heterogeneous blockchains, is largely ignored in those protocols.

In industry, Cosmos [32] and Polkadot [33] 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 [28] for Cosmos and Substrate [34] 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) [35], 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 [36]. 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 [34].

Existing blockchain platforms allow developers to write contracts using new languages such as Solidity [37] and Vyper [38] or a tailored version of the existing languages such as Go, Javascript, and C++. Facebook recently released Move [39], 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 [40, 41], network intrusions [42–44], and enterprise systems [45, 46], 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 CONCLUSION

In this paper, we provided the first generic and measurable definition for Web3.0 based on our observations and analysis of the blockchain infrastructure evolution. Within this definition, we articulate three key infrastructural enablers: individual smart-contract capable blockchains, federated or centralized state publishers, and interoperability platforms to hyperconnect those isolated systems. Then, we presented HyperService, the first interoperability platform usable in the era of Web3.0. 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 over 62,000 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.

References

[1] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, and Y.-C. Hu, "HyperService: Interoperability and Programmability across Heterogeneous Blockchains," in *ACM CCS*, 2019.

[2] A. Back, M. Corallo, L. Dashjr, M. Friedenbach, G. Maxwell, A. Miller, A. Poelstra, J. Timón, and P. Wuille, "Enabling Blockchain Innovations with Pegged Sidechains," *URL: tinyurl.com/mj656p7*, 2014.

[3] M. Herlihy, "Atomic Cross-Chain Swaps," in *ACM PODC*, 2018.

[4] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. Knottenbelt, "XCLAIM: Trustless, Interoperable, Cryptocurrency-Backed Assets," in *IEEE Symposium on Security and Privacy*, 2019.

[5] P. Gazi, A. Kiayias, and D. Zindros, "Proof-of-stake Sidechains," in *IEEE Symposium on Security & Privacy*, 2019.

[6] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, 2014.

[7] "Bitcoin Wiki: Atomic Cross-Chain Trading," https://en.bitcoin.it/wiki/Atomic_swap, Accessed on 2019.

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

[9] I. Eyal, A. E. Gencer, E. G. Sirer, and R. Van Renesse, "Bitcoin-NG: A Scalable Blockchain Protocol," in *USENIX NSDI*, 2016.

[10] J. Wang and H. Wang, "Monoxide: Scale out blockchains with asynchronous consensus zones," in *USENIX NSDI*, 2019.

[11] M. Zamani, M. Movahedi, and M. Raykova, "RapidChain: Scaling Blockchain via Full Sharding," in *ACM CCS*, 2018.

[12] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "OmniLedger: A Secure, Scale-out, Decentralized Ledger via Sharding," in *IEEE Symposium on Security and Privacy*, 2018.

[13] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A Sharded Smart Contracts Platform," *NDSS*, 2017.

[14] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The Blockchain Model of Cryptography and Privacy-preserving Smart Contracts," in *IEEE Symposium on Security and Privacy*, 2016.

[15] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. M. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contract Execution," in *IEEE EuroS&P*, 2019.

[16] J. Krupp and C. Rossow, "teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts," in *USENIX Security Symposium*, 2018.

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

[18] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *ACM CCS*, 2016.

[19] H. Wu, W. Zheng, A. Chiesa, R. A. Popa, and I. Stoica, "DIZK: A Distributed Zero Knowledge Proof System," in *USENIX Security Symposium*, 2018.

[20] F. Zhang, D. Maram, H. Malvai, S. Goldfeder, and A. Juels, "DECO: Liberating web data using decentralized oracles for TLS," in *ACM SIGSAC CCS*, 2020.

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

[22] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town Crier: An Authenticated Data Feed for Smart Contracts," in *ACM CCS*, 2016.

[23] J. Garay, A. Kiayias, and N. Leonardos, "The Bitcoin Backbone Protocol with Chains of Variable Difficulty," in *Annual International Cryptology Conference*, 2017.

[24] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, and Y.-C. Hu, "HyperService: Interoperability and Programmability Across Heterogeneous Blockchains," Cryptology ePrint Archive, Report 2020/578, 2020, https://eprint.iacr.org/2020/578.

[25] R. Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols," in *IEEE Symposium on Foundations of Computer Science*, 2001.

[26] "Open Source Code for HyperService by HyperService-Consortium," https://github.com/HyperService-Consortium, 2019.

[27] T. Parr, "Antlr," https://www.antlr.org/, 2014.

[28] "Tendermint Core," https://tendermint.com, Accessed on 2019.

[29] "Oraclize," http://www.oraclize.it, Accessed on 2019.

[30] "J.P. Morgan: Blockchain and Distributed Ledger," https://www.jpmorgan.com/global/blockchain, Accessed on 2019.

[31] A. Kiayias and D. Zindros, "Proof-of-work Sidechains," Cryptology ePrint Archive, Report 2018/1048, Tech. Rep., 2018.

[32] "Cosmos," https://cosmos.network, Accessed on 2019.

[33] "Polkadot," https://polkadot.network, Accessed on 2019.

[34] "Substrate," https://github.com/paritytech/substrate, Accessed on 2019.

[35] "Standards for the Cosmos network & Interchain Ecosystem." https://github.com/cosmos/ics, Accessed on 2019.

[36] "Cosmos WhitePaper," https://cosmos.network/resources/whitepaper, 2019.

[37] "Solidity," https://solidity.readthedocs.io/en/v0.5.6/, Accessed on 2019.

[38] "Vyper," https://github.com/ethereum/vyper, Accessed on 2019.

[39] S. Blackshear and et al, "Move: A language with programmable resources," The Libra Association, Tech. Rep., 2019.

[40] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat, "Mace: Language support for building distributed systems," in *ACM PLDI*, 2007.

[41] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica, "Declarative networking: Language, execution and optimization," in *SIGMOD*, 2006.

[42] K. Borders, J. Springer, and M. Burnside, "Chimera: A Declarative Language for Streaming Network Traffic Analysis," in *USENIX Security Symposium*, 2012.

[43] R. Sommer, M. Vallentin, L. De Carli, and V. Paxson, "Hilti: An abstract execution environment for deep, stateful network traffic analysis," in *IMC*, 2014.

[44] M. Vallentin, V. Paxson, and R. Sommer, "VAST: A Unified Platform for Interactive Network Forensics," in *USENIX NSDI*, 2016.

[45] P. Gao, X. Xiao, Z. Li, K. Jee, F. Xu, S. R. Kulkarni, and P. Mittal, "AIQL: Enabling Efficient Attack Investigation from System Monitoring Data," in *USENIX ATC*, 2018.

[46] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "SAQL: A Stream-based Query System for Real-time Abnormal System Behavior Detection," in *USENIX Security Symposium*, 2018.

Zhuotao Liu received a Ph.D. from University of Illinois at Urbana-Champaign and a B.S. from Shanghai Jiao Tong University. He is currently an assistant professor of Institute for Network Sciences and Cyberspace, Tsinghua University. Before joining Tsinghua, he was a technical lead at Google, managing massive-scale software-defined datacenter networks. His research interests include network security & privacy, Blockchain infrastructure, datacenter networking and systems security.

Xiangxi Yang is an undergraduate student in Beijing University of Posts and Telecommunications, working on Blockchain Infrastructure.

Shi Jian is a master student in Case Western Reserve University, working on software engineering and programming languages.

Peng Gao is a Postdoctoral Researcher in Computer Science at UC Berkeley. He received his Ph.D. in Electrical Engineering from Princeton University in 2019. His research interest lies in security and privacy issues in systems and networks. His work centers on creating scalable, secure, and trustworthy systems to solve real-world problems.

Haoyu Wang is an associate Professor in the School of Computer Science at Beijing University of Posts and Telecommunications (BUPT). His research covers a wide range of topics in Software Analysis, Privacy and Security, eCrime, Internet/System Measurement, and AI Security. He received his PhD degree in Computer Science from Peking University in 2016.

Xusheng Xiao is an Assistant Professor in the Department of Computer and Data Sciences at Case Western Reserve University. He received his Ph.D. degree from North Carolina State University. Before joining Case Western Reserve University, he worked on software and system security for NEC Labs America. His research interests are software engineering and computer security.

Bihan Wen received the B.Eng. degree in electrical and electronic engineering from Nanyang Technological University, Singapore, in 2012, the M.S. and Ph.D. degrees in electrical and computer engineering from University of Illinois at Urbana-Champaign, USA, in 2015 and 2018, respectively. He is currently a Nanyang Assistant Professor with the School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. His research interests span areas of machine learning, computational imaging, computer vision, image and video processing, and big data applications.

Qi Li received the PhD degree from Tsinghua University. Now he is an associate professor of Institute for Network Sciences and Cyberspace, Tsinghua University. He has ever worked with ETH Zurich and the University of Texas at San Antonio. His research interests include network and system security, particularly in Internet and cloud security, mobile security, and big data security. He is currently an editorial board member of IEEE TDSC and ACM DTRAP.

Yih-Chun Hu is an associate professor in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign, Yih-Chun received my B.S. Magna Cum Laude in 1997 in Computer Science and Mathematics from the University of Washington. Yih-Chun's current research interests are in network security and wireless networks and have published papers in the areas of secure Internet routing, DDoS-resiliant forwarding, secure routing in wireless ad hoc networks, security and anonymity in peer-to-peer networks, efficient cryptographic mechanisms for routing security, and the design and evaluation of multihop wireless network routing protocols, including Quality-of-Service mechanisms for ad hoc networks.