Atomic Smart Contract Interoperability with High Efficiency via Cross-Chain Integrated Execution

Chaoyue Yin, Mingzhe Li, *Member, IEEE*, Jin Zhang, *Member, IEEE*, You Lin, Qingsong Wei, *Senior Member, IEEE*, and Siow Mong Rick Goh, *Senior Member, IEEE*

Abstract*—With the development of Ethereum, numerous blockchains compatible with Ethereum's execution environment (i.e., Ethereum Virtual Machine, EVM) have emerged. Developers can leverage smart contracts to run various complex decentralized applications on top of blockchains. However, the increasing number of EVM-compatible blockchains has introduced significant challenges in cross-chain interoperability, particularly in ensuring efficiency and atomicity for the whole cross-chain application. Existing solutions are *either limited in guaranteeing overall atomicity for the cross-chain application, or inefficient due to the need for multiple rounds of cross-chain smart contract execution.

To address this gap, we propose `IntegrateX`, an efficient cross-chain interoperability system that ensures the overall atomicity of cross-chain smart contract invocations. The core idea is to *deploy the logic required for cross-chain execution onto a single blockchain, where it can be executed in an integrated manner.* This allows cross-chain applications to perform all cross-chain logic efficiently within the same blockchain. `IntegrateX` consists of a *cross-chain smart contract deployment protocol* and a *cross-chain smart contract integrated execution protocol.* The former achieves efficient and secure cross-chain deployment by decoupling smart contract logic from state, and employing an off-chain cross-chain deployment mechanism combined with on-chain cross-chain verification. The latter ensures atomicity of cross-chain invocations through a 2PC-based mechanism, and enhances performance through transaction aggregation and fine-grained state lock. We implement a prototype of `IntegrateX`. Extensive experiments demonstrate that it reduces up to 61.2% latency compared to the state-of-the-art baseline while maintaining low gas consumption.

***Index Terms*—Efficient and Atomic Interoperability, Cross-Chain Smart Contract Invocation, Integrated Execution**

I. Introduction

This work was supported by the National Natural Science Foundation of China (NSFC) under Grant T2495254 and Fang Keng Fellowship.

C. Yin is with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (email: 12432716@mail.sustech.edu.cn).

M. Li is with the School of Computing and Information Technology, Great Bay University, Dongguan 523000, China, and with the Institute of High Performance Computing, A*STAR, Singapore (email: mlibn@connect.ust.hk).

J. Zhang is with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China, and also with Peng Cheng Laboratory, Shenzhen 518055, China (email: zhangj4@sustech.edu.cn).

Y. Lin is with the Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen 518055, China (email: liny2021@mail.sustech.edu.cn).

Q. Wei and S. Goh are with the Institute of High Performance Computing (IHPC), Agency for Science, Technology and Research (A*STAR), Singapore (email: wei_qingsong@ihpc.a-star.edu.sg, gohsm@ihpc.a-star.edu.sg).

C. Yin and M. Li are the co-first authors.

J. Zhang is the corresponding author.

WITH the advent of Bitcoin and Ethereum [1], [2], we have witnessed the emergence of an increasing number of programmable blockchains [3], [4]. On these programmable blockchains, developers can write and deploy smart contracts to build various complex decentralized applications (dApps, such as DeFi, NFT, etc. [5], [6]). Among these programmable blockchains, those that share compatible smart contract execution environment with Ethereum (i.e., Ethereum Virtual Machine, a.k.a. EVM [7]) dominate the landscape, accounting for over 90% of the total value locked [8]. With the increasing number of EVM-compatible blockchains and the diversity of dApps running on each chain, the demand for cross-chain dApps has grown significantly [9]. Cross-chain dApps refer to dApps that require ***cross-chain smart contract invocations*** (***CCSCI***) and coordinated execution across multiple blockchains [10].

However, ensuring overall atomicity for the entire cross-chain dApp while efficiently handling CCSCI remains a critical challenge. Consider a classic train-and-hotel problem [11], as illustrated in Figure 1 left. A user wants to book an outbound train ticket (on Train Chain) through a travel agency (on Agency Chain), then book a hotel (on Hotel Chain), followed by a return train ticket (again on Train Chain). The user wants to ensure that the entire series of CCSCI either all succeed or all fail, ensuring overall atomicity. More importantly, maintaining efficiency during CCSCI is crucial. Key considerations include minimizing latency, reducing gas consumption (i.e., monetary cost), and improving concurrency during the CCSCI process by avoiding blocking mechanisms such as global locking. More cross-chain dApp scenarios are discussed in Section VIII.

Existing CCSCI interoperability solutions generally *either fail to ensure the overall atomicity of a cross-chain dApp or guarantee overall atomicity but with low efficiency.* To ensure atomicity in the CCSCI process, a widely adopted approach is to use a *two-phase commit (2PC) mechanism [12], [10] involving locking and unlocking*. For example, some works [13], [14], [15], [16], [17], [18], [19], [20] propose general cross-chain communication protocols that facilitate the transfer of information and data between blockchains through bridging smart contracts deployed on each blockchain. However, these approaches typically ensure at best the atomicity of single-step cross-chain interactions (Figure 1, a single arrow), but fail to guarantee overall atomicity for the entire cross-chain dApp. Some other recent studies attempt to explore how to ensure overall atomicity for the cross-chain dApp [21], [22], [23], [10]. To ensure overall atomicity, the relevant states must remain locked throughout the entire CCSCI process. However,

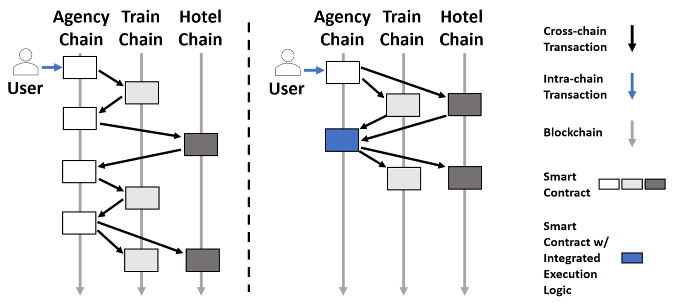


Fig. 1: An example of existing CCSCI solutions (left) and IntegrateX (right) in the Train-and-Hotel scenario.

they commonly face challenges in achieving efficiency. The main reason is that, these approaches usually require *multiple rounds of cross-chain execution and cross-chain information transfer* when handling a cross-chain dApp (Figure 1, left), since a cross-chain dApp usually involves interdependent execution logic distributed over multiple blockchains. It is evident that this method tends to be time-consuming and inefficient (as the locking time could be long), especially when dealing with complex CCSCI. More related work (e.g., cross-chain asset swap, smart contract portability) is discussed and differentiated in detail in Section II.

To fill the research gap, we propose `IntegrateX`, an ***efficient** interoperability system that guarantees **overall atomicity*** for the cross-chain dApp across EVM-compatible blockchains. To enhance the efficiency of CCSCI, our core idea is to *clone and deploy*[1] *the logic of all contracts involved in a cross-chain dApp—originally distributed across multiple chains—onto a single blockchain*. This chain thus integrates the entire execution logic of the cross-chain dApp. When the CCSCI is required, this chain can perform ***integrated execution** in one transaction* for all related logic, after receiving the necessary states (Figure 1, right). Since multi-round cross-chain executions and interactions are no longer required, `IntegrateX` maintains high efficiency, even in complex cross-chain dApps.

However, the design of `IntegrateX` faces several challenges.

Challenge 1: *How to efficiently and securely deploy smart contracts across chains*. Blindly copying full contracts (logic + state) across chains is gas-intensive and vulnerable to byte-code tampering. We address this by proposing an on-chain/off-chain ***Hybrid Cross-Chain Smart Contract Deployment Protocol***. Specifically, we *decouple logic from state*, clone only the logic contracts to one target chain (a.k.a. execution chain), and keep state on their original chains. An *off-chain byte-code transfer* cuts gas costs, while an *on-chain hash comparison* between the source and target contracts cryptographically proves correctness.

Challenge 2: *How to achieve high-concurrency, low-overhead during CCSCI while preserving overall atomicity*. During a cross-chain dApp call, multiple states on different chains must be locked, updated, and released as one atomic unit—without significantly throttling performance. We meet this requirement with a 2PC based ***Cross-Chain Smart Contract Integrated Execution Protocol***. It (i) locks only the relevant *fine-grained state* slices, (ii) *aggregates multiple cross-chain state transfer* into a single round trip, and (iii) *executes the entire dApp call in one transaction* before unlocking. This design sustains concurrency and keeps latency and gas low even for deep call graphs.

[1]The logic on the original chain still exists and continues to function normally.

This paper mainly makes the following contributions:

- We present `IntegrateX`, a cross-chain interoperability system that efficiently facilitates CCSCI while ensuring overall atomicity for the cross-chain dApp. `IntegrateX` can be flexibly deployed on EVM-compatible blockchains without requiring modifications to the underlying blockchain systems.
- In `IntegrateX`, we propose the Hybrid Cross-Chain Smart Contract Deployment Protocol. It achieves efficient and secure cross-chain deployment through the decoupling of smart contract logic and state, and the hybrid approach of off-chain logic deployment and on-chain comparison verification.
- In `IntegrateX`, we propose the Cross-Chain Smart Contract Integrated Execution Protocol. It ensures overall atomicity of cross-chain invocations through a 2PC-based atomic integrated execution mechanism, and enhances the protocol efficiency through an aggregated cross-chain transaction transmission mechanism and fine-grained state lock.
- We implement a prototype of `IntegrateX` and make it open source [24]. Extensive experiments based on real-world use cases demonstrate that `IntegrateX` reduces latency by up to 61.2% meanwhile maintains low gas cost and high concurrency compared to the state-of-the-art baseline. In more complex cross-chain invocations, `IntegrateX` will further improve efficiency.

The remainder of this paper is organized as follows. Section II reviews related work on CCSCI solutions. Section III presents the system model and architecture. Section IV elaborates on the Hybrid Cross-Chain Smart Contract Deployment Protocol. Section V describes the details of the Cross-Chain Smart Contract Integrated Execution Protocol. Section VI analyzes the security of the proposed system. Section VII evaluates the performance of `IntegrateX`. Section VIII discusses the limitations and future directions. Finally, Section IX concludes the paper.

II. Background and Related Work

Cross-Chain Asset Swap and Transfer. Blockchain interoperability has gained significant attention. One widely studied area is cross-chain asset swap and transfer [25], [26], [27], [28], [29], [30], [31], [32], [33], [34], [35]. Cross-chain asset swap protocols enable untrusted parties to exchange assets across different blockchain networks, while cross-chain asset transfer protocols involve locking or burning assets on the source chain and creating their equivalents on the target chain. While these protocols provide atomicity guarantees during cross-chain asset swaps or transfers, their protocols cannot be applied to our target scenario of CCSCI.

Interoperability Protocol for General Communication. Besides cross-chain asset transfers and swaps, some other efforts aim to provide interoperability protocols for more general cross-chain communication [13], [14], [15], [16], [17], [18], [19], [20], enabling the transfer of general data beyond assets between blockchains. However, the primary limitation of these protocols is that they typically guarantee atomicity for, at most, a single cross-chain step between two blockchains. For general cross-chain dApps, which often require multiple rounds of cross-chain interactions across multiple chains, these protocols fail to ensure atomicity for the entire cross-chain dApp. In contrast to these approaches, `IntegrateX` *guarantees overall atomicity of the entire multi-round CCSCI for general cross-chain dApps.*

CCSCI with Overall Atomicity. There are some studies that attempt to ensure overall atomicity for the cross-chain dApp with complex CCSCI [36], [37], [38], [39], [40], [21], [22], [23]. However, these approaches have various limitations. For example, HyperService [36] only can provide weak atomicity, named financial atomicity, via an insurance mechanism. It fails to provide strong execution atomicity guarantees during CCSCI. Some works like Heterogeneous Paxos [40] provide overall atomicity for CCSCI. However, they usually require the underlying blockchains to make targeted modifications to fit their protocols, limiting their generality. Some other works, like Unity [37], also ensure atomicity for a whole CCSCI process. However, their protocols only apply to cross-chain scenarios between two blockchains and does not take into account the more general multi-chain case.

There are few other solutions that guarantee overall atomicity for the cross-chain dApp with complex CCSCI, as shown in TABLE I. However, they generally suffer from inefficiency [21], [22], [23]. A complex cross-chain dApp typically involves multiple blockchains and several interdependent application logic components (implemented through smart contracts). In these works, handling such complexity requires multiple rounds of cross-chain execution and cross-chain interaction in sequential order across several involved chains. Specifically, this involves locking relevant states, executing the logic fragment, reaching consensus, transferring intermediate states to the next blockchain responsible for executing the subsequent logic fragment, and finally unlocking and updating the states. As a result, they typically suffer from high latency and poor concurrency handling CCSCI. For example, Robinson et al. introduced GPACT [21] to achieve atomic cross-chain transactions for Ethereum-based blockchains. However GPACT need to lock entire contracts during updates, reducing availability and performance. Additionally, during the execution of cross-chain applications, GPACT requires sequentially waiting for each smart contract call to be executed on its respective blockchain and for the results to be transferred across chains, resulting in significant overhead and high latency. Additionally, Atomic IBC [22] relies on the Cosmos Hub to provide security guarantees for its protocol. This approach is not easily adaptable to other ecosystems (e.g., EVM-compatible blockchains) and may introduce security bottlenecks.

Unlike these aforementioned works, `IntegrateX` is able to maintain high efficiency during CCSCI while ensuring overall atomicity. Moreover, it can be deployed to blockchains with the same execution environment (e.g., EVM-compatible) without requiring modifications to the underlying blockchain.

Smart Contract Portability. Another type of approach, known as smart contract portability [41], [42], [43], involves frequently moving or replicating entire smart contracts with all the states associated across different blockchains. This allows cross-chain invocations to be converted into intra-chain invocations during contract execution. However, the practicality of this approach is often questionable. The choice of which blockchain to deploy and run a smart contract and dApp typically involves several considerations, including security, ecosystem, and business concern. Moving an entire smart contract from one ecosystem to another is often not favored by developers, as it may compromise security or disrupt the completeness of the ecosystem. Besides, smart contracts often manage a significant amount of user state, and frequently relocating all of this state across chains incurs substantial overhead. This inefficiency further reduces the practicality of these approaches.

The philosophy of `IntegrateX` is fundamentally different from smart contract portability. In `IntegrateX`, the states involved in each smart contract are still maintained and updated by their original blockchains. This allows dApps to continue benefiting from the security and ecosystem of their original blockchain for most of the time. During the CCSCI process, we only temporarily lock part of the state relevant to the invocation and transfer it to a single blockchain, where the logic is integrated and executed using that blockchain's execution environment.

III. System Overview

A. System Model and Architecture

Infrastructure Components. `IntegrateX` operates over a network of *existing blockchains* interconnected via *off-chain relayers* and *on-chain bridging contracts*. Figure 2 provides a high-level architecture overview of `IntegrateX`.

In `IntegrateX`, there are n (variable) ***existing blockchains*** that share the same smart contract execution environment (e.g., EVM-compatible in this paper). These blockchains are developed by their respective projects (e.g., Ethereum, BSC, etc. [2], [44]) and operated by their own blockchain nodes, which handle transaction processing and reach consensus within the blockchain.

`IntegrateX` also features m (variable) ***relayers***. Relayers are off-chain agents (implemented as independent processes) responsible for listening to events from bridging contracts and relaying information across chains, similar to many mainstream interoperability protocols [22], [15], [45], [46]. Each relayer holds an account (public–private key pair) on each blockchain, and the relayers do not require special privileges; they interact through standard transactions and digitally sign all relayed messages. This requires maintaining a sufficient gas balance on each chain, which can be monitored and refilled as needed via external scripts or relayer management services.

TABLE I: Comparison of Related Works about CCSCI Execution

	Strong Atomicity	Compatible with Existing Blockchain	EVM-compatible	Single-round Execution
AtomCI[23],Ivyredaction[39],GPACT[21],Unity[37]	✓	✓	✓	✗
Atomic IBC[22]	✓	✓	✗	✗
Heterogeneous Paxos[40]	✓	✗	✗	✓
HyperService[36]	✗	✗	✓	✗
SmartSysc[38]	read-only	✓	✓	✓
IntegrateX	✓	✓	✓	✓

To incentivize honest and timely relaying, our system adopts a fee mechanism: users who initiate cross-chain operations are required to pay a cross-chain fee, which is forwarded to the relayer upon successful message delivery. This compensates the relayer for gas costs and operational overhead, while providing an economic incentive aligned with protocol correctness. This model is inspired by, and aligned with, practices in IBC [15], where relayers are independent yet economically motivated.

Additionally, IntegrateX's ***bridging smart contracts*** are deployed on each blockchain (serve as on-chain light clients, like in [15], [22]). The bridging contracts mainly serve to verify cross-chain transaction proofs, register all smart contracts in IntegrateX that are eligible for cross-chain invocation, and emit events externally to initiate cross-chain actions. This transport layer of IntegrateX (relayers + bridging contracts) is similar to that of existing interoperability protocols (e.g., IBC) [15], [22], and provides basic security (ensuring that only valid, agreed-upon cross-chain messages are accepted). However, IntegrateX achieves efficiency and overall atomicity at the application layer, which these other protocols do not.

Application Layer Components. In addition to infrastructure components, we distinguish several roles in the IntegrateX ecosystem's application layer. ***Intra-chain dApp providers*** are *developers* who create traditional single-chain dApps and deploy the corresponding smart contracts on a blockchain of their choice. These intra-chain providers are not active participants in cross-chain protocols; rather, they supply reusable contract logic that could later be invoked across chains. ***Cross-Chain dApp providers*** are *developers* who compose multiple intra-chain dApps (potentially on different chains) into a cross-chain dApp. A cross-chain dApp typically consists of multiple interrelated smart contracts on different blockchains. The cross-chain provider decides which blockchain will serve as the *execution chain* (i.e., the single target chain that will host the integrated execution for their dApp's cross-chain logic) and which other blockchains will be *invoked chains* supplying state to the execution chain. (In IntegrateX, any EVM-compatible blockchain can be chosen as the execution chain; no custom blockchain is required, as long as the IntegrateX bridging contracts are deployed on it.) Finally, ***end-users*** interact with the deployed cross-chain dApps by sending transactions to one of the blockchains (often the execution chain) to trigger the cross-chain functionality.

B. Threat Model

In IntegrateX, we assume the standard threat model common in cross-chain systems [22], [15], [45], [46], [23], [21]. For each blockchain, the proportion of Byzantine nodes is assumed to be less than the fault tolerance threshold t of the respective blockchain network (e.g., for blockchains using BFT-type consensus protocols under partial-synchronous network, $t = 1/3$ [47]). This ensures the safety and liveness of consensus within each blockchain. For the relayers, we make only minimal trust assumptions, assuming that at least one relayer is honest and functioning correctly. This assumption is consistent with those made in many existing secure interoperability protocols [22], [15], [45], [46]. As for the dApp providers and users—who represent the application layer components—we make no specific threat assumptions, similar to most existing works [15], [23], [21]. However, we do discuss common countermeasures for dealing with malicious behavior from these components in Section VIII.

C. Objective

IntegrateX aims to achieve the following primary objectives:

- **Efficiency**: During the process of cross-chain smart contract deployment and invocation, IntegrateX seeks to reduce latency, lower gas consumption, and increase transaction concurrency.
- **Overall Atomicity**: IntegrateX aims to ensure overall atomicity for cross-chain dApps that require it during CCSCI. This means guaranteeing that the series of CCSCI operations required by cross-chain dApp providers either all succeed or all fail.

In addition, IntegrateX aims to possess the following desirable properties. *Reliability*: IntegrateX ensures that cross-chain transactions can still be completed even in the presence of malicious relayers. *Verifiability*: IntegrateX guarantees that cross-chain transactions can be verified for authenticity, completeness, and validity, even if malicious relayers are involved. *Consistency*: IntegrateX ensures that the state changes across the blockchains involved in the cross-chain transaction remain coordinated and consistent.

The proofs and experiments related to aforementioned properties are detailed in Section VI and Section VII.

D. Overview of the Core Protocols

The operation of IntegrateX consists of two main protocols: *Hybrid Cross-Chain Smart Contract Deployment Protocol*, and *Cross-Chain Smart Contract Integrated Execution Protocol*, as illustrated in Figure 2.

Hybrid Cross-Chain Smart Contract Deployment Protocol. To address the high overhead and complexity associated with

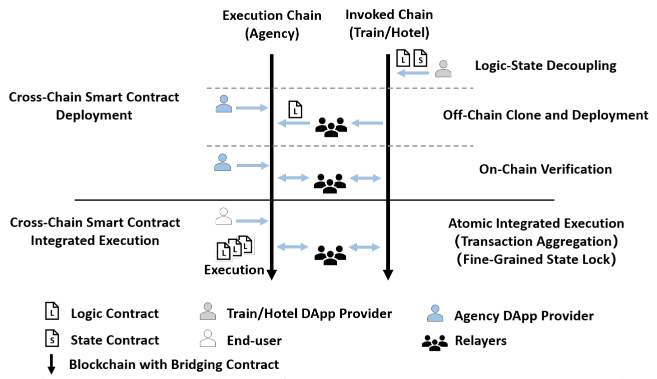

Execution Chain (Agency)
Invoked Chain (Train/Hotel)
Logic-State Decoupling
Cross-Chain Smart Contract Deployment
Off-Chain Clone and Deployment
On-Chain Verification
Cross-Chain Smart Contract Integrated Execution
Atomic Integrated Execution (Transaction Aggregation) (Fine-Grained State Lock)
Execution
Logic Contract
State Contract
Blockchain with Bridging Contract
Train/Hotel DApp Provider
End-user
Agency DApp Provider
Relayers

Fig. 2: The overview of `IntegrateX`'s core protocols.

cross-chain deployment of smart contracts, we design a hybrid deployment protocol that improves efficiency while ensuring security and flexibility. The proposed protocol consists of three core phases: *Logic-State Decoupling*, *Off-Chain Clone and Deployment*, and *On-Chain Verification*, as shown in Figure 2.

The first phase, *Logic-State Decoupling*, tackles the challenge of redundant data migration in cross-chain scenarios. We devise a set of guidelines that enables dApp developers to separate smart contracts into *logic execution contracts* and *state storage contracts*. This approach enables our protocol to clone and deploy only the logic contracts onto the execution chain, while the state-heavy contracts remain in their original locations. In this phase, dApp providers can flexibly develop logic contracts and state contracts according to our defined logic-state decoupling guidelines (Section IV-A).

To further minimize the gas cost and latency involved in cross-chain logic contract deployment, we design an *Off-Chain Clone and Deployment* mechanism. Instead of performing all operations on-chain, which is resource-intensive, our protocol enables the off-chain cloning and deployment of logic contracts onto the execution chain. This phase is triggered only when dApp providers need to deploy logic contracts from one invoked chain onto the execution chain for execution purposes. The execution chain can be chosen flexibly, and a designated transaction can initiate the deployment process. `IntegrateX` implements this off-chain deployment technique (Section IV-B) to optimize cost-efficiency.

However, off-chain operations pose integrity risks due to their susceptibility to tampering. To ensure correctness and trustworthiness, we introduce an *On-Chain Verification* phase. In this phase, bridging smart contracts, which are deployed on both the invoked chain and the execution chain, are used to perform on-chain cross-verification of the logic contract bytecodes between the two chains. This verification mechanism (Section IV-C) guarantees that the deployed contracts are authentic and consistent with their original versions, thereby securing the deployment process against malicious interference.

Through this three-phase protocol design, we effectively reduce deployment overhead while maintaining the integrity and scalability of `IntegrateX`.

Cross-Chain Smart Contract Integrated Execution Protocol. To achieve high-concurrency, low-overhead and guarantee

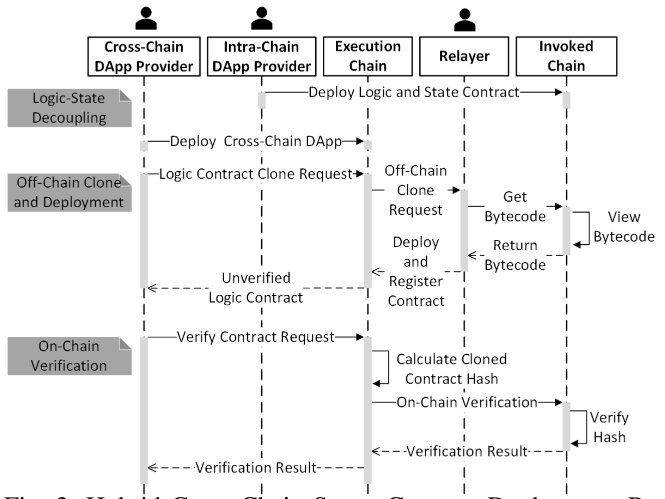


Fig. 3: Hybrid Cross-Chain Smart Contract Deployment Protocol.

atomicity during CCSCI, we design an integrated execution protocol based on the two-phase commit (2PC) paradigm, following mainstream practices (Section V-A). The protocol coordinates execution by first locking the relevant contract states across participating invoked chains, transferring these states to a designated execution chain, executing the combined logic, and finally returning the updated states to invoked chains for unlocking and commitment. This ensures that cross-chain transactions either complete entirely or are rolled back in their entirety, preserving consistency.

However, frequent cross-chain state transfers can introduce substantial communication overhead. To address this challenge, we introduce a transaction aggregation mechanism that consolidates multiple cross-chain interactions into a single transaction when applicable, thereby reducing cross-chain communication costs (Section V-B).

Another key issue is the limited concurrency introduced by state locking. In conventional designs, locking entire contract states can hinder parallel transaction processing. To improve concurrency, our protocol incorporates a fine-grained locking scheme (Section V-C). Specifically, we define a set of development guidelines that enable dApp developers to decompose contract states and apply locks at a finer granularity. By isolating and locking only the necessary sub-states, the protocol enables concurrent execution of unrelated operations, significantly enhancing scalability and throughput in multichain environments.

Overall, the protocol ensures atomicity, reduces communication overhead, and improves concurrency, thereby enabling reliable and efficient execution of cross-chain dApps.

IV. Hybrid Cross-Chain Smart Contract Deployment Protocol

A. Logic-State Decoupling

To efficiently perform CCSCI, we need to deploy the logic of the invoked contracts to the same execution chain for efficient integrated execution. Existing contracts often contain both logic and state, and directly cloning such contracts would result in high gas costs and additional state storage overhead on the execution chain. Therefore, we design a set of Logic-State Decoupling (LSD) Guidelines to guide the developers to decouple existing contracts into logic execution contracts and state storage contracts. During cross-chain deployment, only the logic contracts need to be cloned, which reduces gas costs. Developers can follow these guidelines to develop new contracts with separated logic and state or upgrade existing contracts by decoupling them. We now provide a detailed explanation of the LSD Guidelines. Moreover, a simple example illustrating the LSD is provided in Section IV-A1.

State Contract. According to the LSD Guidelines, the decoupled state contract first includes all the *variables* (representing states) from the original contract, as well as all the *view functions* that read the contract's state. Since view functions are read-only and do not generate transactions, they do not affect cross-chain execution. Additionally, the state contract contains *functions for locking and updating* the contract state, as cross-chain invocations require locking and updating states. The state contract should also contain *functions that call the logic contract* in order to interact normally with the logic contract.

Logic Contract. In the logic contract, no variables are stored. It only contains the functions that implement the original contract's *execution logic*. These functions are called by the state contract to carry out the dApp's logic operations. When the state contract calls these functions, it passes all necessary state data (variables), and after the functions complete execution, the results are returned to the state contract. In our protocol, only the logic contract needs to be cloned, which reduces gas costs during cross-chain deployment.

Remarks. Our protocol also supports existing smart contracts, even if they are not decoupled into logic and state. However, in such cases, the cross-chain deployment process will incur higher gas costs. In this case, the contracts deployed to the execution chain do not actively maintain their own state. Instead, they passively update their state during each cross-chain invocation. The discussion related to developers' learning costs is given in Section VIII.

1) Logic-State Decoupling Example: We now give a sample of logic-state decoupling. In the following code 1, after applying logic-state decoupling, the original `Hotel` contract is split into two separate contracts: the logic contract `LHotel` and the state contract `SHotel`. The `LHotel` contract contains no state variables and only includes the `book()` function, which implements the hotel reservation functionality. Since there are no variables within the `LHotel` contract, the `book()` function must take all necessary parameters as inputs.

On the other hand, the `SHotel` contract retains all the variables from the original Hotel contract and introduces an additional address variable, `addr_lhotel`, which records the address of the `LHotel` contract. In the `book()` function of the `SHotel` contract, no reservation logic is implemented directly; instead, it calls the `book()` function from `LHotel` using the `addr_lhotel` parameter to execute the hotel reservation functionality.

By decoupling the logic and state in this way, only the `LHotel` contract needs to be cloned. Since `LHotel` contains no state variables, this approach significantly reduces gas consumption during the cross-chain clone and deployment process.

```
contract Hotel{
    int256 price;
    int256 remain;
    mapping (address => int256) accounts;
    function getPrice() public view returns(uint256)
        ;
    function getRemain() public view returns(uint256
        );
    function book(address user_addr, uint256 num)
        public returns(uint256);
    function lockState(bytes[] memory args) public
        returns();
    function updateState(bytes[] memory args) public
         returns();

contract LHotel{
    function book(uint256 price, uint256 remain,
        uint256 num) public returns(uint256)
}

contract SHotel{
    int256 price;
    int256 remain;
    address addr_lhotel;
    uint256 lock_size;
    struct lock_bag;
    mapping (uint256 => lock_bag) lockpool;
    mapping (address => int256) accounts;
    function setLocksize(uint256) public returns();
    function getPrice() public view returns(uint256)
        ;
    function getRemain() public view returns(uint256
        );
    function lockState(bytes[] memory args) public
        returns();
    function updateState(bytes[] memory args) public
         returns();
    function book(address user_addr, uint256 num)
        public returns(uint256);
```

Listing 1: Pseudocode of Hotel Logic-State Decoupling

B. Off-Chain Clone and Deployment

Transmitting contract bytecode on-chain (via contract event) requires a significant amount of gas. To improve the efficiency of cross-chain logic deployment and reduce gas consumption, we propose *transferring the contract bytecode via an off-chain solution*. The off-chain clone and deployment consists of two phases: the preparation phase and the clone phase. The main process is shown in Figure 3.

Preparation. In this phase, the cross-chain dApp provider needs to obtain the call tree of smart contracts and determine which logic contracts need to be cloned. Similar to existing works [48], [21], [23], this is achieved by first applying static analysis tools such as Slither [49] to extract the call graph of the application. External contract calls can be identified by parsing the call graph, where calls from one contract to another are marked. Static analysis tools can highlight these relationships, helping developers understand cross-contract dependencies. Therefore, from this call graph, the provider constructs the corresponding call tree, which captures the expected invocation paths during a single cross-chain transaction.

This process enables the identification of all logic contracts (a superset) that must be cloned to the invoked chain in order to ensure correct and atomic execution. After this, the developer can choose a blockchain to deploy the cross-chain dApp. It is important to note that we offer developers a high degree of flexibility: They can select any blockchain according to their preference. For efficiency, the provider may pick a chain that already hosts some required logic contracts so that those do not need to be cloned again. To reduce costs, they may also choose a blockchain that already hosts some required logic contracts, so that no additional relayer fees are incurred for cloning those contracts again. (Since cross-chain fees are ultimately paid by the provider and compensated to the relayers, this ultimately saves the provider's expenses.) Once the selection is made, the developer sends a transaction to invoke the bridging contract on the chosen chain. The bridging contract then triggers an event to notify the relayers to initiate the cross-chain deployment. The event includes the ID of the invoked chain and the addresses of the logic contracts $Addr_{\mathrm{L}}$ to be cloned on the invoked chain. This concludes the preparation phase.

Remarks on Call Tree Analysis. Modern static analysis tools provide reliable extraction of contract call graphs and enable accurate construction of call trees. Even in the presence of misconfiguration or adversarial behavior by a dApp provider, the implications of an incorrect call tree are limited. Since the dApp provider is responsible for covering the gas costs of contract cloning, there is a built-in economic disincentive to over- or under-provision the call tree. An incorrect call tree may result in the cloning of unnecessary contracts on the execution chain, thereby increasing storage usage, but it does not introduce any security risk or compromise the atomicity of the cross-chain protocol. Or, a missing logic clone would lead to a runtime exception on the execution chain, effectively aborting the cross-chain transaction. However, our protocol remains atomically safe (no partial commits occur on other chains, but the dApp's operation would fail). The call tree is not stored on the execution chain. Instead, it is constructed off-chain by the dApp provider during the preparation phase and is used solely as input to the bridging contract when initiating the cross-chain deployment. Only the necessary information—such as the addresses of the logic contracts to be cloned and the identifier of the invoked chain—is packaged and passed as parameters to the bridging contract. This design minimizes on-chain storage and computation overhead, while still providing relayers with the necessary context to coordinate cross-chain deployment.

Clone. After the event is triggered, relayers will detect the event and use the `getcode()` function from the bridging contract on the invoked chain to obtain the bytecode of the contract that needs to be cloned. This process is an *off-chain read-only inquiry and, therefore, does not consume gas.* Subsequently, the relayers will obtain the ABI file which defines the contract interface. The relayers will then deploy the contract to the execution chain. Once a relayer completes the redeployment, it registers the address of the cloned logic contract $Addr'_{\mathrm{L}}$, through the bridging contract on the execution chain, marking the end of the clone phase.

Remarks on Clone Reliability. For the cross-chain reliability of the protocol, multiple relayers perform the clone process after detecting the bridging contract's event. Therefore, even if some relayers do not respond to the event, other relayers will ultimately complete the contract clone and deployment. Once one relayer completes the deployment, the bridging contract will trigger an event to stop the others, ensuring that the logic contract will not be deployed multiple times. Furthermore, when a contract is cloned on the execution chain, the deployment transaction will invoke the *register* function of the bridging contract deployed on the same chain. The bridging contract maintains a mapping from a service identifier (`string`) to the corresponding logic contract address. Only the first registration for a given identifier will succeed. The deployment transaction will revert if the *register* fails. This means the contract will not be created on the blockchain, and no bytecode will be stored. This mechanism prevents multiple cloning attempts, as any subsequent invocation of *register* will fail, causing the entire cloning transaction to fail. As a result, only the first cloning attempt can successfully deploy the contract on the execution chain.

Moreover, if a malicious or overly eager relayer attempts to clone a logic contract without the presence of an event trigger, it must bear the gas cost independently. Relayers are thus economically disincentivized from performing premature cloning, as no reimbursement is provided in the absence of an authorized event. Even if such premature cloning occurs, it does not compromise system correctness; the resulting contracts are merely redundant and remain unused, leading at most to minor storage overhead.

C. On-Chain Verification

Although off-chain clone and deployment can achieve efficient cross-chain logic deployment, it does not guarantee security as there may be malicious relayers present in the system. A malicious relayer could potentially modify the contract bytecode or deploy a wrong contract. To ensure verifiability and security during the off-chain clone and deployment process, we propose the on-chain cross-chain verification scheme to *compare the cloned logic contract with the original contract and verify its correctness*. The on-chain verification process is shown in Figure 3.

After the cross-chain deployment is completed, the cross-chain dApp provider can initiate cross-chain verification on the execution chain by calling the `Verification()` function of the bridging contract. The bridging contract will search the cloned contract bytecode of address $Addr'_{\mathrm{L}}$, and calculate the hash of the bytecode. Then the bridging contract triggers an event includes the hash computation result. The relayers are responsible for transmitting this information (a cross-chain transaction) along with its *receipt proof* and *block header* to the invoked chain. The *receipt proof* serves as a cryptographic proof that a specific event (e.g., a function execution or asset transfer) has indeed occurred on the source chain. It typically includes a Merkle proof [50], which attests that the corresponding event log is embedded in a finalized block. The *block header* provides the *receiptsRoot*, which serves as the cryptographic commitment to the receipt set of the block and is used to verify the inclusion of the event log via the receipt proof. This enables the invoked chain to verify the authenticity of the event without relying on trust assumptions about the relayer, thereby ensuring the integrity of cross-chain execution. In the invoked chain, the receipt proof with the cross-chain transaction is first verified to ensure the result has already reached consensus on the execution chain. The bridging contract then searches the corresponding local contract bytecode based on the address $Addr_{\mathrm{L}}$, and calculates the hash of the bytecode. Then, the bridging contract will verify whether it matches the hash transmitted across the chain. The result of the verification is then returned.

The bridging contract on the execution chain receives the verification result from the invoked chain. To ensure authenticity, the execution chain first verifies that the provided *block header* originates from the invoked chain and is finalized (e.g., via the bridging contract). Then, using the accompanying *receipt proof*, it reconstructs the Merkle path to the target receipt and verifies that the computed *receiptsRoot* matches the one in the provided *block header*. This process guarantees the integrity and authenticity of the event data emitted on the invoked chain. If the result confirms that the bytecode of the cloned contract on the execution chain matches the original, the bridging contract marks the cloned contract as verified. Only verified contracts are permitted to participate in subsequent cross-chain invocations. Upon successful verification, the relayer responsible for the deployment is rewarded. If the verification fails, the relayer will be penalized, and the off-chain clone and deployment process will be restart.

Remarks. For the reliability of the cross-chain protocol, multiple relayers could transmit the same cross-chain transaction. However, the bridging contracts will deduplicate identical transactions from multiple relayers to avoid multiple executions on-chain. This process also applies in the subsequent integrated execution protocol.

V. Cross-Chain Smart Contract Integrated Execution Protocol

A. Atomic Integrated Execution

To efficiently and atomically execute complex CCSCI, we propose an atomic integrated execution scheme. As all the invoked logic has been migrated onto the execution chain, the atomic integrated execution does not need multiple rounds of cross-chain execution when handling CCSCI. This enables all the logic to be executed within a single transaction, enhancing the efficiency of CCSCI. Moreover, to ensure overall atomicity for a series of CCSCI, we employ a state synchronization mechanism based on the Two-Phase Commit protocol [12]. This process locks the relevant states across the involved chains, transmits the states to the chain responsible for integrated execution, and then returns the states to the respective chains to unlock and update them after the execution is completed. The entire atomic integrated execution consists Locking, Integrated Execution and Updating, which is shown in the Figure 4.

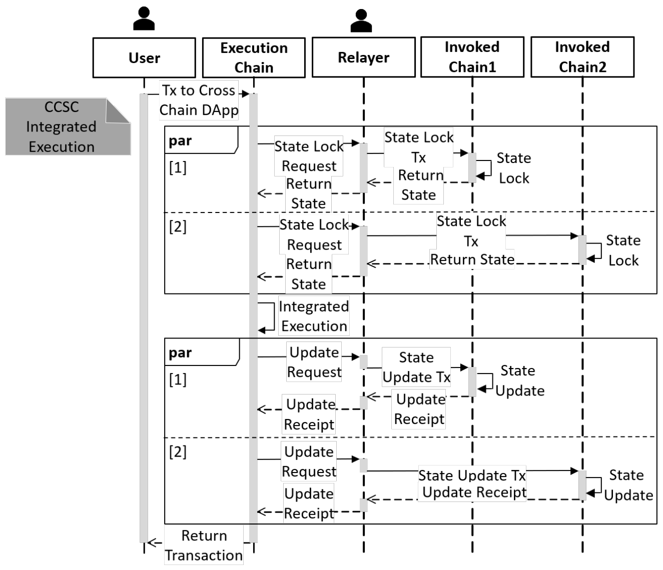


Fig. 4: `IntegrateX`'s Cross-Chain Smart Contract Integrated Execution Protocol.

Locking. The Locking process locks all the required invoked contract states on the invoked chain. The cross-chain dApp provider can obtain all the required state on the call tree beforehand via tools such as static analysis [49]. Based on this information, a user sends a transaction via the cross-chain dApp to invoke the cross-chain dApp contract. The cross-chain dApp contract then calls the bridging contract to issue an event to lock the relevant states on the invoked chains. When relayers detect the event, it will transfer this message (i.e., cross-chain transaction with Merkle proof) to each invoked chain. The bridging contract on the invoked chain will verify the authenticity of the cross-chain though calculating the Merkle proof of the transaction and invoke the `LockState()` function of each invoked contract. Once the bridging contract has successfully locked the state and retrieved the required contract states, it will trigger an event to return the states. After the relayers detect the event, they will transmit these states via cross-chain transactions to the execution chain. After the execution chain's bridging contract verifies the authenticity of the transactions via the Merkle proof, it will return the states to the dApp. Once all requested states are returned from individual invoked chains, the Locking process ends, and these states are used as inputs for the Integrated Execution.

Integrated Execution. The Integrate Execution process executes the entire CCSCI logic on the execution chain. Cross-chain dApp contracts on the execution chain use the requested state values as inputs to perform the full call tree execution. Since all contracts required for the cross-chain invocation have completed logic migration and have been verified already, the integrated execution can be completed within a single transaction on the execution chain. The cross-chain dApp contract records the output results of each invoked contract during the Integrated Execution, allowing for state updates of the invoked contracts on other chains after the execution is completed. Once execution is complete, the Integrated

Execution ends and transitions to the Updating process.

Updating. The Updating process unlocks and updates all the invoked contract states on the invoked chains. After Integrated Execution is completed on the execution chain, the cross-chain dApp contract triggers an event to update the result via bridging contract. The relayers will distribute the result to the invoked chains. The bridging contract on each invoked chain verifies the Merkle proof of the cross-chain transaction and then invokes the `UpdateState()` function of each invoked contract, which will unlock and update the states of the invoked contracts. It is important to note that in `IntegrateX`, the address of the bridging contract is a required parameter during the initialization of a state contract. In the *state contract*, an `address` variable is used to store the address of the *bridging contract*. Before executing any function related to state locking or updating, the contract verifies that the caller is the registered bridging contract. Only upon successful verification are the locking or updating operations permitted. This design allows the state contract to grant the bridging contract the permission to invoke specific internal functions, thereby enabling it to accept state locking and state updates initiated by the bridging contract.

Remarks: ***Rollback.*** During the Atomic Integrated Execution, state rollback may occur due to failure in locking the state or execution failure. The invoked state might already be locked by another cross-chain invocation, which causes the failure to lock the invoked state. In this case, the bridging contract on the execution chain will initiate a new event to unlock all associated contracts that have already been locked and returned in this cross-chain invocation. For the contracts that have not yet completed the locking process, the event will cancel the lock attempt, thereby ensuring overall atomicity. The execution might also fail, due to the reason such as insufficient gas fee or insufficient states. In this case, the bridging contract on the execution chain will discard the obtained states and initiate a cross-chain event to unlock all locked contracts, thus ensuring all invoked contracts are either fully locked or not locked at all.

For security concerns, within CCSCI, only the bridging contract is authorized (configured in the contract logic) to unlock the state contracts, while the relayer has no direct access to unlock any locked contract state. Furthermore, the relayer is incapable of forging an unlock request to the bridging contract, since any maliciously crafted message cannot pass the Merkle proof verification and will therefore be rejected during contract execution. This design ensures the security and integrity of CCSCI execution. Besides, the rollback process does not involve any state updates. All pending state changes reside on the execution chain, and rollback merely discards the result of the integrated execution on the execution chain while releasing the locked states on the invoked chains. As such, `IntegrateX` does not risk unintended consequences for other CCSCI.

Timeout. Additionally, to prevent the invoked contract state from being locked for an extended period, a design of timeout will be determined by the dApp developer within the application and managed by the execution chain. Timeout is a widely adopted mechanism in lock-based solutions [25], [22]. In `IntegrateX`, both the bridging contract (set by our protocol) and the developer-defined smart contract can specify a maximum timeout. During execution, the effective timeout is determined by taking the minimum of these two values, which prevents the dApp provider from setting an indefinite timeout period. Moreover, timeout in blockchain systems is typically measured in terms of block depth rather than wall-clock time. Therefore, although the actual block time may vary across different blockchains, the maximum timeout in `IntegrateX` can be aligned approximately by using chain-specific block depths. For example, blockchains such as Ethereum[2] tend to have longer block intervals (e.g., 12 seconds), so a timeout of 10 blocks would translate to roughly 2 minutes. In contrast, blockchains such as BNB Smart Chain [51] may have shorter block times (e.g., 2-3 seconds), in which case a timeout of 10 blocks would correspond to a much shorter wall-clock duration. Moreover, the timeout is typically set to exceed the finality confirmation time of the underlying blockchain for safety. For transactions that time out, such as when the locking is not completed or execution is not finished for an extended period, the execution chain will mark the cross-chain call as failed and send a cross-chain event to unlock the relevant contracts.

Finality. In `IntegrateX`, all cross-chain transactions must wait until the consensus on the initiating chain is finalized (or, for Nakamoto-type consensus [1], highly likely to be finalized) before being committed to the execution chain or invoked chain, to ensure cross-chain security. For different blockchains, `IntegrateX` allows the configuration of varying confirmation depths to ensure that the result of a transaction is sufficiently finalized. Specifically, it supports setting a block *confirmation threshold* in the bridging contract, enabling tailored finality strategies for heterogeneous blockchains.

To ensure that only finalized states are used on the execution/invoked chain, the bridging contract verifies the confirmation status of the submitted block header from the invoked/execution chain. It compares the submitted header with the latest block height of the invoked/execution chain and ensures that the block has reached the required confirmation depth. If this condition is not satisfied, the cross-chain message will be rejected (can be resend). In addition, the bridging contract checks the validity of the message through the *receipt proof*. Only when both the *confirmation threshold* and *receipt proof* verification succeed will the event be accepted and its associated state be used for further execution. For different blockchains, the threshold setting is typically related to how long it takes for the blockchain itself to reach finality. Information regarding finality is usually defined in their project documentations. `IntegrateX` only needs to set the corresponding threshold in the bridging contract according to the definition in their documentations before connecting to their blockchains. This flexible confirmation mechanism ensures that cross-chain execution is performed based on finalized and reliable states.

B. Transaction Aggregation

A large amount of cross-chain transactions for transmitting state incurs significant gas consumption. Handling CCSCI

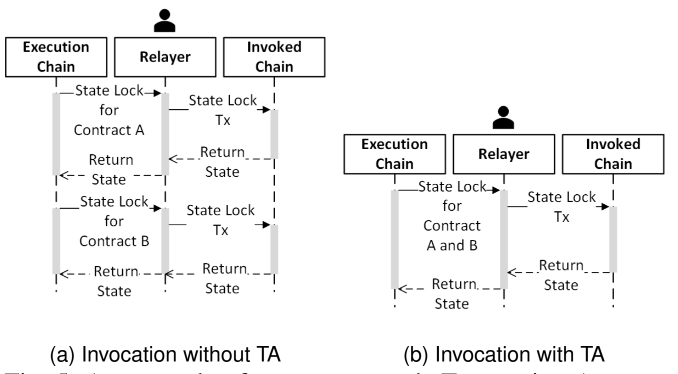


Fig. 5: An example of `IntegrateX`'s Transaction Aggregation mechanism.

by sequential invocation may need to invoke contracts on the same chain in multiple rounds, which cause *multiple rounds of cross-chain state transfers*. To address this issue, we design a transaction aggregation mechanism to reduce gas costs caused by multiple invocations of different contracts and state transfers on the same chain.

Our protocol locks all required states simultaneously, allowing multiple states on the same chain to be locked together, even when the calls are non-contiguous. The transaction aggregation mechanism combines all state requests on the same chain into a single transaction, reducing the number of cross-chain transactions. Similarly, during state updates, the transaction aggregation mechanism reduces the number of update requests. Therefore, for cross-chain calls involving multiple contracts on the same chain, this approach ensures that the number of cross-chain transactions is equal to the number of invoked chains (rather than the number of contracts), thereby reducing gas consumption.

Figure 5 shows an example that the execution chain needs to lock the state of both *Contract A* and *Contract B* on the invoked chain. Sequential invocation needs to invoke *Contract A* and *Contract B* in multiple rounds, as illustrated in Figure 5a. By using transaction aggregation mechanism, the execution chain only needs to send a single transaction to lock the state of both *Contract A* and *Contract B*, as shown in Figure 5b. Through this mechanism, the execution chain can avoid multiple rounds of cross-chain messaging, improving efficiency. Furthermore, the reduction in the number of transactions also leads to lower gas consumption.

C. Fine-Grained State Lock

During the atomic integrate execution process, we need to lock contract states to ensure atomicity. A simple approach is to either lock the entire state of the invoked contract or lock individual states being invoked [23], [21]. However, these state-locking mechanism reduces transaction concurrency. Because once a state is locked, any subsequent transactions related to that state will fail until the state is unlocked. Therefore, we establish a set of guidelines to guide developers in decomposing certain states that can be split into finer granularity, and allowing *partial state locking*. Unlike existing protocols that require locking the entire state, our fine-grained state lock mechanism locks only partial of a state at a fine-grained level. This approach enhances concurrency by reducing unnecessary state locking.

In EVM-compatible blockchains, Solidity is the smart contract language. In Solidity, the state of a contract is typically represented by *variables*. Various types of variables are used in Solidity, such as `uint`, `address`, and `boolean`. We find that the `uint` variable is widely used and can be decomposed. Based on this observation, we design a fine-grained state lock specifically for `uint` variables and develop a lock pool mechanism. The lock pool is a structure where, during the use of the fine-grained state lock, part of the state is locked within this structure until execution is completed, while the unlocked portion of this state remains accessible, thereby enhancing the concurrency of the application.

For variables that can be directly derived from transaction inputs, the exact value of the required state can be precisely determined. A fine-grained state lock can be applied to accurately lock only the relevant portion of such state. For states that are dynamically used during execution, their exact values cannot be determined at the beginning. We allow dApp developers to lock these states in fixed-size increments based on their needs.

For example, as shown in Listing 1 in Section IV-A1, users can book hotel rooms through a smart contract. For simplicity, we assume all rooms are of the same type, and the *remain* variable represents the number of available rooms. Without fine-grained state locking, a booking attempt by one user would lock the entire contract state, preventing concurrent transactions. In the *SHotel* contract, a fine-grained locking mechanism is implemented using a *lock_bag* structure, which allows partial locking of the *remain* variable. This enables other transactions to access and interact with the remaining state. During the locking period, a *lockpool* temporarily holds the locked portion of the state. Developers can configure the *lock_size* parameter to determine how much of the state is locked per transaction, helping to prevent failures caused by insufficient locked state.

Remarks. We focus on the flexibility of this mechanism, allowing developers to choose whether to implement the finer granularity locking mechanism in their dApp and to set the lock granularity based on the specific needs of the dApp. Moreover, developers can set the fixed size of the fine-grained state lock based on their preferences, tailored to the specific use case of the application. More discussion related to developers' learning costs is given in Section VIII.

VI. Security Analysis

A. Security in Hybrid Cross-Chain Smart Contract Deployment Protocol

Theorem 1. *The Hybrid Cross-Chain Smart Contract Deployment Protocol ensures reliability, verifiability, and consistency, if the proportion of Byzantine nodes in each blockchain is less than its fault tolerance threshold, and at least one functional relayer is present.*

Proof. **Reliability.** The Hybrid Cross-Chain Smart Contract Deployment Protocol ensures reliability, meaning that when an execution chain issues a request for cross-chain deployment of a contract, the requested contract will eventually be deployed to the execution chain. Within the Hybrid Cross-Chain Smart Contract Deployment Protocol, multiple relayers listen for such requests. Even if malicious relayers intentionally ignore the requests, assuming that at least one functional relayer exists, this relayer will ultimately handle the logic clone and deployment of the requested contract. Thus, even in the worst-case scenario, the protocol ensures that the contract will be successfully deployed to the execution chain.

Verifiability. The Hybrid Cross-Chain Smart Contract Deployment Protocol ensures verifiability, meaning that both the execution chain and the invoked chain are able to verify the cross-chain transactions transmitted by the relayers. Additionally, both chains can verify the hash values of the contract bytecode before and after the cross-chain clone and deployment to ensure that the contract has been deployed correctly.

In the Hybrid Cross-Chain Smart Contract Deployment Protocol, multiple relayers listen to the request and relay the messages. The invoked chain (bridging contract) validates the authenticity of the transactions using the Merkle proof attached with the cross-chain transactions, thereby preventing the relayers from altering the cross-chain data. By parsing the transactions, the invoked chain can obtain the bytecode hash of the cloned contract and compare it with the bytecode hash of the original contract on the chain to ensure the correctness of the cross-chain deployment. Additionally, the nonce value associated with each transaction prevents malicious replay attacks.

It is worth noting that the concept of a relayer is not originally proposed by `IntegrateX`, nor is the assumption of at least one honest relayer unique to this system. In fact, all existing cross-chain protocols rely on the presence of at least one genuinely honest and non-compromised relayer [21], [22], [23], [36], [39]. This relayer must not be forgeable or impersonated by an adversary; otherwise, the threat model would be fundamentally violated and the protocol's security guarantees would no longer hold. During cross-chain communication, the relayer solely acts as a message carrier. It listens to on-chain events and forwards the corresponding message along with a *receipt proof* (mentioned in Section IV-C), which attests to the inclusion of the event in the block header of the source chain. Upon receiving the message, the invoked chain verifies the *receipt proof* to ensure the message was indeed generated by the execution chain. Only after successful verification does the *bridging contract* on the invoked chain reconstruct and execute the intended contract call. This verification mechanism is symmetric and also applies to messages sent from the invoked chain back to the execution chain, thereby ensuring the integrity of cross-chain communication regardless of relayer trustworthiness. Therefore, when inconsistencies arise due to a relayer maliciously modifying the transmitted message, the tampered data will be rejected by the invoked chain as it fails the on-chain proof verification. Only messages that pass the verification will be accepted.

Furthermore, in the case of blockchain forks leading to conflicting relayer messages, `IntegrateX` defers execution until one fork becomes finalized (mentioned in Section V-A). Once finality is reached, the corresponding transaction results are adopted for further processing.

Consistency. The Hybrid Cross-Chain Smart Contract Deployment Protocol ensures consistency, meaning that during the off-chain clone and deployment as well as the on-chain verification process, both the execution chain and the invoked chain reach agreement on the outcome of the cross-chain requests. In the Hybrid Cross-Chain Smart Contract Deployment Protocol, the proportion of malicious nodes on both the execution chain and the invoked chain remains within the fault tolerance threshold. Moreover, the protocol requires to wait until consensus on one chain is finalized (or highly likely to be finalized) before committing the cross-chain transaction to another chain. As a result, even in the presence of Byzantine nodes, each chain can still achieve consensus on cross-chain transactions and finalize them, ensuring that consistency is not undermined by malicious nodes. This guarantees that the outcomes of cross-chain requests remain consistent. □

B. Security in Cross-Chain Smart Contract Integrated Execution Protocol

Theorem 2. *The Hybrid Cross-Chain Smart Contract Integrated Execution Protocol ensures overall atomicity, reliability, verifiability, and consistency, if the proportion of Byzantine nodes in each blockchain is less than its fault tolerance threshold, and at least one functional relayer is present.*

Proof. **Overall Atomicity.** The Cross-Chain Smart Contract Integrated Execution Protocol guarantees overall atomicity, which means that in the selected CCSCI process, state changes on both the execution chain and the invoked chain either all succeed or all fail, preventing any situation where one chain's state changes while the other does not. We employ the atomic integrated execution mechanism, similar to the 2PC scheme. For one cross-chain dApp, during this process, all the states required by the invocation on the invoked chains will be locked. If any contract has already been locked by another invocation, this invocation will fail, and all other invoked contracts will be unlocked. If the execution chain obtains all the necessary states but the execution fails due to insufficient gas or other reasons, the execution will be aborted, and all related locked contracts will be unlocked without any state changes. Furthermore, the protocol incorporates a timeout scheme: If any of the invoked chains fails to return the required state within the specified time frame by dApp, or if the execution transaction on the execution chain fails to complete within the specified time limit, the execution chain will abort the invocation and unlock all related contract states, ignoring any subsequent state responses from the invoked chains. As a result, only when the execution chain has successfully acquired all required states and completed execution will it issue state updates to all related invoked chains, thereby ensuring the atomicity of the entire CCSCI process.

It is important to note that, similar to existing cross-chain solutions, `IntegrateX` incorporates an acknowledgment (receipt) mechanism for cross-chain messaging [52], [22]. In the

Updating phase of integrated execution, the execution chain receives receipts from all invoked chains. The final result is output on the execution chain only after all receipts confirm that the state updates have been successfully completed on the respective chains. This mechanism ensures that the execution chain is informed whether the updating states transaction has been successfully executed. Only upon successful execution will the system proceed to the next step. Any failure in the cross-chain execution triggers a full rollback of the transaction, thereby preserving the overall atomicity of `IntegrateX`. As a result, even if an update operation on the invoked chain fails (which barely happens), the entire transaction will be rolled back, maintaining the system's atomicity guarantees.

Reliability. The Cross-Chain Smart Contract Integrated Execution Protocol ensures reliability, which means that when an execution chain initiates a CCSCI request, the invoked chain will eventually receive the cross-chain transaction, and the state returned by the invoked chain will likewise be received by the execution chain. In the Cross-Chain Smart Contract Integrated Execution Protocol, multiple relayers monitor CCSCI requests. Even in the presence of malicious relayers who deliberately fail to respond to the request, the assumption of at least one functional relayer ensures that, in the worst-case scenario, this relayer will relay the cross-chain transaction to the invoked chain, ensuring that the transaction is eventually received. Similarly, even in the worst-case scenario, at least one relayer will transmit the state returned by the invoked chain back to the execution chain, thereby guaranteeing the reliability of the CCSCI process.

Verifiability. The Cross-Chain Smart Contract Integrated Execution Protocol ensures verifiability, which means the ability of the invoked chain to verify the authenticity of cross-chain transactions transmitted by relayers from the execution chain, while the execution chain can also verify the transactions returned by the invoked chain. In this protocol, multiple relayers listen for the request and relay messages. Both the execution chain and the invoked chain can independently validate the authenticity of the cross-chain transactions using the Merkle proof attached with the transactions. Additionally, the use of transaction nonce values prevents malicious replay attacks, ensuring the integrity of the cross-chain interaction.

Consistency. The Cross-Chain Smart Contract Integrated Execution Protocol guarantees consistency, ensuring that both the execution chain and the invoked chain agree on the result of the requested operation during a cross-chain smart contract invocation. In this protocol, the proportion of malicious nodes on both chains remains within the fault tolerance threshold, allowing consensus to be achieved even if there exist Byzantine nodes. Additionally, the protocol requires waiting until consensus on one chain has been finalized before committing the cross-chain transaction to the other chain. Therefore, cross-chain transactions on the blockchain cannot be maliciously altered, ensuring that both the execution chain and the invoked chain reach a unified agreement on the outcome of cross-chain operations. □

VII. Implementation and Evaluation

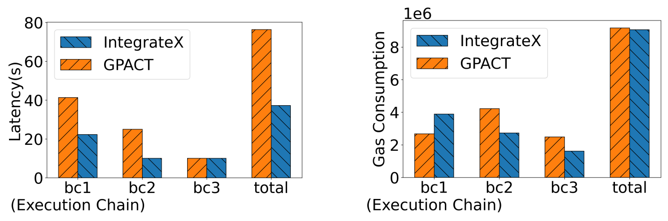
A. Implementation

We implement a prototype of `IntegrateX` based on a cross-chain communication project Bitxhub [52] (its transport layer is similar to IBC). The relayers are implemented in Golang, while the bridging contracts are implemented using Solidity. For the underlying blockchain, we do not rely on a custom-built chain. Instead, we leverage the existing `go-ethereum` client to implement several EVM-compatible blockchains in Golang, without making any modifications to the underlying blockchain infrastructure. For performance comparison, we also implement a baseline CCSCI protocol GPACT [21]. As mentioned in the paper, GPACT is one of the most advanced existing protocols that can guarantee overall atomicity for complex CCSCI. However, GPACT has limitations in efficiency, as it requires multiple rounds of cross-chain execution and message exchange. Furthermore, we use Solidity to implement the Train-and-Hotel example mentioned in the paper, along with smart contracts in various other scenarios, to demonstrate the `IntegrateX`'s performance in real use cases and under different conditions.

For the ease of implementation and experimentation, the underlying blockchain employs a Proof-of-Authority (PoA) mechanism for leader selection, and achieves consensus through Practical Byzantine Fault Tolerance (PBFT). PoA is a consensus protocol in which a limited set of pre-approved validators, referred to as authorities, are responsible for block production.

To support our proposed protocols, we implement a bridging contract on the blockchain. The protocol requires no modification to the underlying blockchain infrastructure; it only involves deploying these bridging contracts. We implement the functions *regServer* and *regState* to register the logic contract and the state contract, respectively. During the on-chain verification process, the *compareBytes* function in the bridging contract is used to compare the bytecode of the cloned contract with that of the original contract on the invoked chain, ensuring that the contract has been correctly cloned. In the Cross-Chain Smart Contract Integrated Execution Protocol, the bridging contract is also responsible for issuing the CCSCI transaction. To this end, we implement the *lockState* and *updateState* functions to lock and update the contract state on the invoked chain. Importantly, this integration is lightweight and non-intrusive, requiring no changes to the underlying blockchain beyond smart contract deployment.

Furthermore, `IntegrateX` supports cross-chain interoperability via a relayer built upon BitXHub's cross-chain communication framework. The relayer includes a receipt mechanism to ensure the reliability and traceability of message delivery. Unlike traditional cross-chain systems that rely on trusted relayers for message validation, the relayer in `IntegrateX` is entirely trustless. Its sole responsibility is to relay raw cross-chain data—such as events, receipts, and Merkle proofs—between chains. All critical verification, including block header validation, Merkle proof checking, and event authenticity verification, is executed on-chain through smart contracts. This design eliminates the need to trust any



(a) Latency (b) Gas consumption

Fig. 6: Latency and gas consumption in the Train-and-Hotel use case.

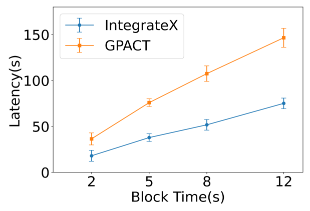
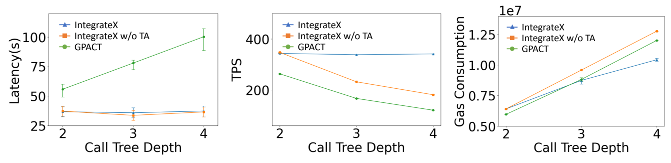


Fig. 7: Latency of handling CCSCI in `IntegrateX` with different block times.



(a) Latency (b) Throughput (c) Gas consumption

Fig. 8: Latency, gas consumption and throughput under different call tree depths.

off-chain component and ensures that incorrect or forged messages submitted by a malicious relayer will be rejected by the verification logic on the execution chain. Consequently, the relayer can be treated as a permissionless data courier, significantly enhancing the security and decentralization of the system.

B. Experimental Setup

We test the performance of `IntegrateX` on multiple servers. Each server is equipped with an Intel® Core(TM) i5-10400F CPU @ 2.90 GHz and 15.5 GB RAM running 64-bit Ubuntu 22.04 LTS. We deploy 4 relayers via Docker containers on multiple servers. We also deploy 3 blockchains, each with 4 nodes. To decide block time, we notice that existing well-known EVM-compatible blockchains (e.g., BNB Smart Chain [51], Polygon [53], Optimism [54], Ethereum [2]) typically adopt block times ranging from 2 to 12 seconds. As a result, the default block time in the experiments is set to be 5 seconds to maintain generality while ensuring a reasonable balance between performance and realism. We set each block to contain up to 4096 transactions. Additionally, the gas calculation method in the experiments is the same as that used in Ethereum. For every experiment, we select Blockchain 1 (bc1) as the execution chain in the `IntegrateX` system, with the other chains as invoked chains.

C. Experimental Results

1) Performance under Train-and-Hotel Use Case: We now compare the latency and gas consumption during the Train-and-Hotel CCSCI process for `IntegrateX` and GPACT.

`IntegrateX` decreases total latency by 51.2% compared to GPACT, as shown in Figure 6a,. On bc1 (execution chain), the latency of `IntegrateX` is reduced by 46.2% compared to GPACT. The reason is that, due to the integrated execution, `IntegrateX` could integrate and execute all the related execution logic within one block, avoiding multiple rounds of cross-chain execution during CCSCI. On bc2, `IntegrateX` also reduces the latency by 60% compared to GPACT. This is because the train contract is invoked multiple times, requiring multiple rounds of state transfers in GPACT, whereas in `IntegrateX`, only a single round of state transfer is needed.

The results in Figure 6b show that the total gas consumption of `IntegrateX` and GPACT is nearly identical in the Train-and-Hotel use case. On bc1, the gas consumption in `IntegrateX` is larger than GPACT (41.9%). This is mainly because in `IntegrateX`, the execution chain (bc1) handles all the logic during CCSCI, leading to a higher gas. However, on the invoked chains (bc2 and bc3), `IntegrateX` achieves lower gas cost compared to GPACT (35.4%, 35.2%, respectively). The main reason is that in `IntegrateX`, the invoked chains only need to lock and update states without executing any logic.

Current EVM-compatible blockchains exhibit a range of block times. The results presented in Figure 7 show the experiments conducted on the Train-and-Hotel use case under different block time settings. The results indicate that `IntegrateX` consistently demonstrates significantly lower latency compared to GPACT across all block times. This improvement is primarily due to `IntegrateX`'s ability to avoid multiple rounds of cross-chain execution during CCSCI, thereby reducing overall latency. As the block time increases, the latency of both `IntegrateX` and GPACT exhibits an approximately linear growth trend.

2) Performance under Different Invocation Complexity: We evaluate the performance of latency, throughput and gas consumption of `IntegrateX` and GPACT with different CCSCI complexity. We vary the call tree depth to represent different levels of complexity in cross-chain invocations.

The latency results are shown in Figure 8a. Due to the atomic integrated execution mechanism, the latency of `IntegrateX` and `IntegrateX` without transaction aggregation (TA) remain stable at around 35 seconds. The latency of GPACT, however, increases linearly as the complexity of CCSCI grows. Specifically, when the call tree depth is 4, `IntegrateX` significantly reduces latency by 61.2% compared to GPACT. Furthermore, as can be inferred, with invocation complexity further increases, `IntegrateX` will show greater improvements compared to GPACT.

The throughput results are shown in Figure 8b. `IntegrateX` achieves significantly higher throughput than GPACT across varying call tree depths. Specifically, when the call tree depth is 2, 3, and 4, `IntegrateX` outperforms GPACT by 30.4%, 104.3%, and 199.1%, respectively. This improvement stems from `IntegrateX`'s

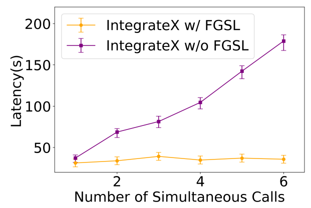


Fig. 9: Latency of handling CCSCI in `IntegrateX` with-/without Fine-Grained State Lock (FGSL).

transaction aggregation mechanism, which ensures that the number of transactions required to execute CCSCI remains constant, regardless of call tree depth. As a result, the throughput of `IntegrateX` remains nearly stable. In contrast, `IntegrateX` without transaction aggregation, as well as GPACT, both exhibit a linear increase in the number of required transactions as the call tree depth increases, leading to a corresponding linear decrease in throughput. Moreover, since integrated execution in `IntegrateX` requires fewer transactions than GPACT even without aggregation, it consistently achieves better throughput. Therefore, the performance advantage of `IntegrateX` becomes increasingly pronounced with deeper call trees.

The gas consumption results are shown in Figure 8c. Due to the lower invocation complexity, the gas savings from transaction aggregation are not significant, when the call tree depth is 3, the gas consumption for CCSCI in `IntegrateX` is nearly the same compared to GPACT. As the invocation complexity increases, when the call tree depth reaches 4, transaction aggregation reduces gas consumption in `IntegrateX` by 9.8% compared to GPACT. And with further increases in invocation complexity, transaction aggregation will reduces more gas consumption in `IntegrateX`. We also evaluate the gas consumption between `IntegrateX` and `IntegrateX` without transaction aggregation. The experimental results show that as the complexity of CCSCI increases, the transaction aggregation mechanism significantly reduces gas consumption in CCSCI. When the call tree depth is 3, the transaction aggregation mechanism reduces gas consumption by 5.4%, and when the call tree depth is 4, it reduces gas consumption by 19.3%. This suggests that the transaction aggregation mechanism achieves greater improvements in gas consumption as the complexity of CCSCI increases.

3) Concurrency Performance with Fine-Grained State Lock: We now measure the concurrency performance of `IntegrateX` with and without the fine-grained state lock mechanism. Concurrency refers to the system's ability to execute multiple full cross-chain smart contract invocations in parallel, rather than isolated operations on a single chain. To show the concurrency performance, we initiate multiple cross-chain calls on the same state simultaneously and compare the time consumed by the cross-chain dApp. The experimental results are shown in the Figure 9.

The experimental results show that with fine-grained state lock mechanism, the latency of `IntegrateX` remains stable at around 35 seconds regardless of the number of simultaneous calls. This demonstrates that the fine-grained state lock can maintain good transaction concurrency. On the other side, as the number of simultaneous calls increases, the latency increases linearly in `IntegrateX` without fine-grained state lock. When the number of simultaneous invocations reached 6, the average latency of the `IntegrateX` system rises to 178.8 seconds, representing an increase of approximately 400.1%.

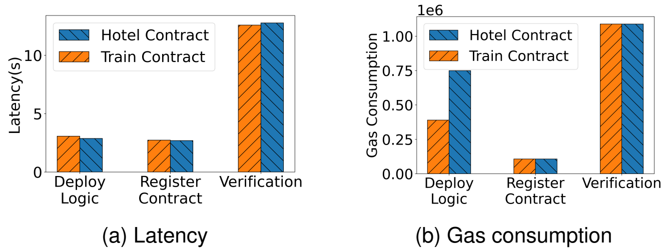


Fig. 10: Latency and gas consumption of different on-chain operations during off-chain clone and deployment.

4) Logic-State Decoupling: We evaluate the gas consumption during the cross-chain smart contract deployment, using the Train-and-Hotel contracts as an example. We compare the results of the train and hotel contracts with and without logic-state decoupling. The experimental results in Table II indicate that, after applying logic-state decoupling, gas consumption during off-chain clone and deployment is reduced by 48.6% for the train contract and 74.5% for the hotel contract. The main reason is that, the train contract is implemented with more complex logic, while the hotel contract stores more states. As can been seen, for contracts with substantial state but relatively simple logic, logic-state decoupling can significantly reduce gas consumption.

TABLE II: Gas consumption during cross-chain deployment.

Contract Type	w/o LSD (Gas)	w/ LSD (Gas)
Train Contract	1,459,626	749,383
Hotel Contract	1,524,151	389,383

5) On-chain Operation During Off-chain Clone and Deployment: We evaluate the gas consumption and latency associated with the on-chain operations during the off-chain clone and deployment phase of the Hybrid Cross-Chain Smart Contract Deployment Protocol. The results are shown in Figure 10. Figure 10a presents the latency of different operations, which are nearly identical for both the *Hotel* and *Train* contracts. Because both the Deploy Logic and Register Contract operations complete within a single on-chain transaction, their latency remains low. Verification, by contrast, requires cross-chain validation and therefore incurs higher latency. Figure 10b shows the gas consumption of these operations. The *Train* contract consumes more gas for logic deployment due to its more complex functionality. However, for contract registration and verification, both contracts perform similar operations, leading to nearly identical gas usage. It is important to note that these operations are one-time costs. Overall, the results demonstrate that the on-chain overhead is reasonably lightweight and practical in terms of both gas and latency.

VIII. Discussions and Limitations

Potential Application Scenarios. Beyond the example of the train-and-hotel problem mentioned in this paper, IntegrateX can be extended to a wide range of application scenarios. One promising use case is cross-chain flash loans [55]. Flash loans are atomic, uncollateralized lending protocols that allow users to borrow funds at nearly zero cost, perform other operations, and then repay the loan. However, these processes require the guarantee of overall atomicity, meaning that either all steps succeed or they all fail. Due to this requirement for atomicity, existing flash loan protocols are limited to intra-chain operations. With IntegrateX, which provides overall atomicity for cross-chain dApps, users can efficiently perform cross-chain flash loan operations. Other promising application scenarios include, but are not limited to, cross-chain atomic arbitrage and cross-chain supply chain management.

Learning Cost for Developers. The logic-state decoupling and fine-grained state lock mechanisms may slightly increase the development learning curve for smart contract developers. Fortunately, we have proposed a set of guidelines to assist developers, and the mechanisms are flexible (as discussed in Section IV-A and V-C), allowing developers to freely decide whether to implement them. Furthermore, we can provide formal documentation, SDKs, and other resources (following existing standards such as Wormhole [56] and IBC [15]) to guide developers in secure and efficient development and auditing, thereby reducing the learning curve. Additionally, incentive mechanisms (e.g., token rewards) are widely adopted in the industry to encourage developers to build and utilize our system. In future research, we could even explore AI-based semi-automated smart contract tools to further address this challenge.

Moreover, logic-state decoupling offers several additional benefits. For instance, it facilitates modular programming principles in smart contract development, which helps reduce subsequent upgrade and maintenance costs while improving contract security. When an issue arises in a specific contract module, developers can conduct targeted audits and resolve vulnerabilities efficiently.

Support for Heterogeneous Chains. IntegrateX currently supports blockchains that run different consensus protocols but share the same smart contract execution environment. As mentioned in the paper, to ensure cross-chain transaction security across blockchains with different consensus protocols, IntegrateX waits until consensus on the source chain is finalized (or highly likely to be finalized) before committing the cross-chain transaction to the target chain. Additionally, while this paper focuses on IntegrateX's implementation on EVM-compatible blockchains, it can also be modified to operate between non-EVM-compatible blockchains that share the same smart contract execution environment.

However, a current limitation of IntegrateX is that it cannot operate between blockchains with different smart contract execution environments. Fortunately, this issue could potentially be addressed using advanced techniques such as code virtualization [57]. Expanding IntegrateX to support integrated execution across blockchains with different smart contract environments is a future research direction we aim to explore.

Trade-Off Between Flexibility and Load Balancing. In IntegrateX, cross-chain dApp providers have the flexibility to select any chain as the execution chain for integrated execution, based on their preferences. However, this flexibility may introduce a potential issue: if many cross-chain dApp providers choose the same chain as the execution chain, that chain could become a hotspot, potentially degrading its performance. One possible solution is for a third party (e.g., IntegrateX) to manage load balancing by selecting the execution chain on behalf of the cross-chain dApp providers. However, this approach could introduce centralization risks. Additionally, as discussed in the paper, developers' choice of which chain to run their dApps on often involves considerations beyond performance, such as ecosystem compatibility and business partnerships. How to better achieve load balancing and how to strike a trade-off between performance and flexibility are important questions that warrant future research.

Mitigating Malicious Application Layer Components. In public blockchain scenarios, there are common strategies to mitigate malicious behavior from application layer components (e.g., dApp providers, users). For instance, a malicious cross-chain dApp provider might attempt to maliciously lock certain states to prevent their usage by others. Such behavior can be countered using contract-based authorization or blacklisting mechanisms (widely used in existing dApp development [58]). For example, an intra-chain dApp provider can pre-arrange with a cross-chain dApp provider and authorize trusted cross-chain dApp providers (through their associated addresses) in their contracts, allowing only authorized cross-chain dApp providers to invoke and lock their states. Similarly, intra-chain dApp providers can blacklist specific cross-chain dApp providers in their contracts to block their interactions. Additionally, gas fee mechanisms can serve as a deterrent to malicious application layer components attempting to launch flooding attacks against the blockchain.

Cross-Chain vs. Cross-Shard. Some existing works have explored the issue of cross-shard smart contract handling [59], [48]. However, cross-chain and cross-shard scenarios are fundamentally different, and their solutions cannot be directly applied to cross-chain contexts. The primary reasons are as follows: First, research on blockchain sharding typically involves modifications to the underlying system [60], [61]. In contrast, a key requirement of cross-chain protocols or systems is that they must not require modifications to the underlying blockchains, ensuring better compatibility with existing blockchain systems. Second, a blockchain sharding system usually has a beacon chain [62], [63] responsible for coordinating progress across shards, which is impractical in cross-chain protocols. After all, in cross-chain scenarios, each blockchain essentially belongs to a different system. Finally, in cross-chain contexts, blockchains are often heterogeneous (e.g., different consensus protocols), which is uncommon in blockchain sharding systems.

Inter-Chain Shared Security. In IntegrateX, each blockchain is assumed to have a proportion of malicious nodes lower than its fault tolerance threshold (i.e., they are secure). This is a widely accepted assumption in most existing works. However, recent research has begun exploring how to achieve secure cross-chain interoperability protocols in scenarios where individual blockchains may not be secure, by sharing security across multiple blockchains [45]. IntegrateX could adopt similar ideas through modifications to achieve shared security among blockchains. However, how to design and implement inter-chain shared security within IntegrateX while still maintaining efficiency is an important direction for future research.

IX. Conclusion

In this paper, we introduce IntegrateX, a system that enhances cross-chain interoperability by ensuring overall atomicity and efficiency in cross-chain smart contract invocations across EVM-compatible blockchains. The proposed hybrid cross-chain smart contract deployment protocol enables the logic contract to be cloned and deployed onto a single blockchain efficiently and securely. The proposed cross-chain smart contract integrated execution protocol allows efficient intra-chain integrated execution for cross-chain smart contract invocations while guarantee overall atomicity. Experimental results show that, compared to the state-of-the-art baseline, IntegrateX is able to reduce a significant portion of latency while maintaining low gas cost and high concurrency, particularly excelling in complex cross-chain interactions.

References

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

[2] V. Buterin *et al.*, "Ethereum white paper," *GitHub repository*, vol. 1, pp. 22–23, 2013.

[3] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A survey on blockchain interoperability: Past, present, and future trends," *ACM Comput. Surv.*, pp. 1–41, 2021.

[4] H. Huang, W. Kong, S. Zhou, Z. Zheng, and S. Guo, "A survey of state-of-the-art on blockchains: Theories, modelings, and tools," *ACM Comput. Surv.*, vol. 54, no. 2, pp. 44:1–44:42, 2021.

[5] W. Li, J. Bu, X. Li, H. Peng, Y. Niu, and Y. Zhang, "A survey of defi security: Challenges and opportunities," *Journal of King Saud University - Computer and Information Sciences*, vol. 34, no. 10, Part B, pp. 10 378–10 404, 2022.

[6] M. Nadini, L. Alessandretti, F. Di Giacinto, M. Martino, L. M. Aiello, and A. Baronchelli, "Mapping the nft revolution: market trends, trade networks, and visual features," *Scientific reports*, vol. 11, no. 1, p. 20902, 2021.

[7] Ethereum, "Ethereum virtual machine (evm) documentation," https://ethereum.org/en/developers/docs/evm/, 2024, accessed: Feb. 2025.

[8] DefiLlama, "Total value locked all chains," https://defillama.com/chains, 2024, accessed: Feb. 2025.

[9] W. Ou, S. Huang, J. Zheng, Q. Zhang, G. Zeng, and W. Han, "An overview on cross-chain: Mechanism, platforms, challenges and advances," *Computer Networks*, vol. 218, p. 109378, 2022.

[10] G. Falazi, U. Breitenbücher, F. Leymann, and S. Schulte, "Cross-chain smart contract invocations: A systematic multi-vocal literature review," *ACM Comput. Surv.*, vol. 56, no. 6, pp. 1–38, 2024.

[11] V. Buterin, "What is the train-and-hotel problem?" https://vitalik.eth.limo/general/2017/12/31/sharding_faq.html\#what-is-the-train-and-hotel-problem, 2017, accessed: Feb. 2025.

[12] B. Lampson and D. Lomet, "A new presumed commit optimization for two phase commit," in *19th International Conference on Very Large Data Bases (VLDB'93)*, 1993, pp. 630–640.

[13] M. Nissl, E. Sallinger, S. Schulte, and M. Borkowski, "Towards cross-blockchain smart contracts," in *2021 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. IEEE, 2021, pp. 85–94.

[14] G. Wood, "Polkadot: Vision for a heterogeneous multi-chain framework," *White paper*, vol. 21, no. 2327, pp. 2327–4662, 2016.

[15] J. Kwon and E. Buchman, "Cosmos white paper," White Paper, 2019.

[16] E. Abebe, D. Behl, C. Govindarajan, Y. Hu, D. Karunamoorthy, P. Novotny, V. Pandit, V. Ramakrishna, and C. Vecchiola, "Enabling enterprise blockchain interoperability with trusted data transfer (industry track)," in *Proceedings of the 20th International Middleware Conference Industrial Track*, 2019, p. 29–35.

[17] M. Darshan, M. Amet, G. Srivastava, and J. Crichigno, "An architecture that enables cross-chain interoperability for next-gen blockchain systems," *IEEE Internet of Things Journal*, vol. 10, no. 20, pp. 18 282–18 291, 2023.

[18] D. Reijsbergen, A. Maw, J. Zhang, T. T. A. Dinh, and A. Datta, "Demo: Piechain - a practical blockchain interoperability framework," in *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*, 2023, pp. 1021–1024.

[19] B. C. Ghosh, T. Bhartia, S. K. Addya, and S. Chakraborty, "Leveraging public-private blockchain interoperability for closed consortium interfacing," in *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, 2021, pp. 1–10.

[20] A. Garoffolo, D. Kaidalov, and R. Oliynykov, "Zendoo: a zk-snark verifiable cross-chain transfer protocol enabling decoupled and decentralized sidechains," in *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, 2020, pp. 1257–1262.

[21] P. Robinson and R. Ramesh, "General purpose atomic crosschain transactions," in *2021 3rd Conference on blockchain research & applications for innovative networks and services (BRAINS)*. IEEE, 2021, pp. 61–68.

[22] AtomicIBC, "Atomic ibc," https://informal.systems/blog/atomic-ibc, 2024, accessed: Feb. 2025.

[23] Y. Chen, A. Asheralieva, and X. Wei, "Atomci: A new system for the atomic cross-chain smart contract invocation spanning heterogeneous blockchains," *IEEE Transactions on Network Science and Engineering*, vol. 11, no. 3, pp. 2782–2796, 2024.

[24] "Implementation of integratex," https://github.com/B-Above/INtegrateX, 2024.

[25] L. Lys, A. Micoulet, and M. Potop-Butucaru, "Atomic swapping bitcoins and ethers," in *2019 38th Symposium on Reliable Distributed Systems (SRDS)*, 2019, pp. 372–3722.

[26] J. Xu, D. Ackerer, and A. Dubovitskaya, "A game-theoretic analysis of cross-chain atomic swaps with htlcs," in *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, 2021, pp. 584–594.

[27] X. Luo, K. Xue, Q. Sun, and J. Lu, "Crosschannel: Efficient and scalable cross-chain transactions through cross-and-off-blockchain micropayment channel," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–15, 2024.

[28] Y. Manevich and A. Akavia, "Cross chain atomic swaps in the absence of time via attribute verifiable timed commitments," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*, 2022, pp. 606–625.

[29] H. Tian, K. Xue, X. Luo, S. Li, J. Xu, J. Liu, J. Zhao, and D. S. Wei, "Enabling cross-chain transactions: A decentralized cryptocurrency exchange protocol," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 3928–3941, 2021.

[30] M. Herlihy, "Atomic cross-chain swaps," in *Proceedings of the 2018 ACM symposium on principles of distributed computing*, 2018, pp. 245–254.

[31] A. Deshpande and M. Herlihy, "Privacy-preserving cross-chain atomic swaps," in *International Conference on Financial Cryptography and Data Security*. Springer, 2020, pp. 540–549.

[32] S. A. Thyagarajan, G. Malavolta, and P. Moreno-Sanchez, "Universal atomic swaps: Secure exchange of coins across all blockchains," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 1299–1316.

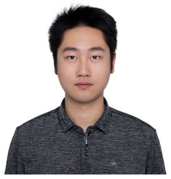
[33] Y. Chen, J.-N. Liu, A. Yang, J. Weng, M.-R. Chen, Z. Liu, and M. Li, "Pacdam: Privacy-preserving and adaptive cross-chain digital asset marketplace," *IEEE Internet of Things Journal*, vol. 11, no. 8, pp. 13 424–13 436, 2024.

[34] L. Yin, J. Xu, and Q. Tang, "Sidechains with fast cross-chain transfers," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 6, pp. 3925–3940, 2022.

[35] A. Zamyatin, D. Harz, J. Lind, P. Panayiotou, A. Gervais, and W. Knottenbelt, "Xclaim: Trustless, interoperable, cryptocurrency-backed assets," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 193–210.

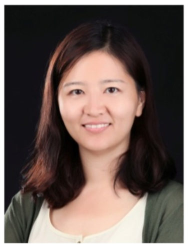
[36] Z. Liu, Y. Xiang, J. Shi, P. Gao, H. Wang, X. Xiao, B. Wen, Q. Li, and Y.-C. Hu, "Make web3. 0 connected," *IEEE transactions on dependable and secure computing*, vol. 19, no. 5, pp. 2965–2981, 2021.

[37] Y. Tao, B. Li, and B. Li, "On atomicity and confidentiality across blockchains under failures," *IEEE Transactions on Knowledge and Data Engineering*, pp. 766–780, 2024.
[38] M. Westerkamp and A. Küpper, "Instant function calls using synchronized cross-blockchain smart contracts," *IEEE Transactions on Network and Service Management*, vol. 20, no. 3, pp. 2136–2150, 2023.
[39] S. Hu, M. Li, J. Weng, J.-N. Liu, J. Weng, and Z. Li, "Ivyredaction: Enabling atomic, consistent and accountable cross-chain rewriting," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 3883–3900, 2024.
[40] "Heterogeneous paxos and multi-chain atomic commits," https://anoma.net/blog/heterogeneous-paxos-and-multi-chain-atomic-commits, 2024, accessed: Feb. 2025.
[41] M. Westerkamp, "Verifiable smart contract portability," in *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2019, pp. 1–9.
[42] E. Fynn, A. Bessani, and F. Pedone, "Smart contracts on the move," in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2020, pp. 233–244.
[43] M. Westerkamp and A. Küpper, "Smartsync: Cross-blockchain smart contract interaction and synchronization," in *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, 2022, pp. 1–9.
[44] B. S. Chain, "Binance smart chain whitepaper," https://github.com/bnb-chain/whitepaper/blob/master/WHITEPAPER.md, accessed: Feb. 2025.
[45] P. Sheng, X. Wang, S. Kannan, K. Nayak, and P. Viswanath, "Trustboost: Boosting trust among interoperable blockchains," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1571–1584.
[46] E. N. Tas, R. Han, D. Tse, and M. Yu, "Interchain timestamping for mesh security," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1585–1599.
[47] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI 99)*, 1999.
[48] M. Li, Y. Lin, J. Zhang, and W. Wang, "Jenga: Orchestrating smart contracts in sharding-based blockchain for efficient processing," in *Proceedings of the 42nd International Conference on Distributed Computing Systems (ICDCS 22)*, 2022.
[49] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2019, pp. 8–15.
[50] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Conference on the Theory and Application of Cryptographic Techniques.*, 1987, pp. 369–378.
[51] B. Chain, "Binance Chain Whitepaper," https://github.com/bnb-chain/whitepaper/blob/master/WHITEPAPER.md, accessed: Jun. 18, 2025.
[52] S. Ye, X. Wang, C. Xu *et al.*, "Bitxhub: a heterogeneous blockchain interoperability platform based on sidechain relaying [j]," *Computer Science*, vol. 47, no. 6, pp. 294–302, 2020.
[53] J. M. Keil, "Polygon decomposition." *Handbook of computational geometry*, vol. 2, pp. 491–518, 2000.
[54] C. S. Carver, M. F. Scheier, and S. C. Segerstrom, "Optimism," *Clinical psychology review*, vol. 30, no. 7, pp. 879–889, 2010.
[55] M. Tefagh, F. Bagheri, A. Khajehpour, and M. Abdi, "Atomic bonded cross-chain debt," in *Proceedings of the 2020 3rd International Conference on Blockchain Technology and Applications*, ser. ICBTA '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 50–54.
[56] Wormhole, "Wormhole docs," https://wormhole.com/docs/, 2024, accessed: Feb. 2025.
[57] ChainsAtlas, "Chainsatlas virtualizatiom unit," https://www.chainsatlas.com/, 2024, accessed: Feb. 2025.
[58] Ethereum, "Etherscan," https://etherscan.io, 2024, accessed: Feb. 2025.
[59] X. Qi and Y. Li, "Lightcross: Sharding with lightweight cross-shard execution for smart contracts," in *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024, pp. 1681–1690.
[60] M. Li, Y. Lin, J. Zhang, and W. Wang, "Cochain: High concurrency blockchain sharding via consensus on consensus," in *Proceedings of the 42nd IEEE Conference on Computer Communications (INFOCOM 23)*, 2023.
[61] M. Li, Y. Lin, W. Wang, and J. Zhang, "Sp-chain: Boosting intrashard and cross-shard security and performance in blockchain sharding," *IEEE Internet of Things Journal*, vol. 12, no. 15, pp. 31 737–31 753, 2025.
[62] M. Li, W. Wang, and J. Zhang, "Lb-chain: Load-balanced and low-latency blockchain sharding via account migration," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 10, pp. 2797–2810, 2023.
[63] Y. Lin, M. Li, and J. Zhang, "Spiralshard: Highly concurrent and secure blockchain sharding via linked cross-shard endorsement," *IEEE Transactions on Networking*, pp. 1–16, 2025.



Chaoyue Yin is currently a master candidate with Department of Computer Science and Engineering, Southern University of Science and Technology. He received his B.E. degree in computer science and technology from Southern University of Science and Technology in 2024. His research interests are mainly in blockchain sharding and interoperability protocol.

Mingzhe Li is currently a US-equivalent Assistant Professor with the School of Computing and Information Technology, Great Bay University. He received his Ph.D. degree from the Department of Computer Science and Engineering, Hong Kong University of Science and Technology in 2022. Prior to that, he received his B.E. degree from Southern University of Science and Technology. His research interests are mainly in blockchain sharding, consensus protocol, blockchain application, network economics, and crowdsourcing.



Jin Zhang is currently an associate professor with Department of Computer Science and Engineering, Southern University of Science and Technology. She received her B.E. and M.E. degrees in electronic engineering from Tsinghua University in 2004 and 2006, respectively, and received her Ph.D. degree in computer science from Hong Kong University of Science and Technology in 2009. Her research interests are mainly in mobile healthcare and wearable computing, wireless communication and networks, network economics, cognitive radio networks and dynamic spectrum management.

You Lin is currently a master candidate with Department of Computer Science and Engineering, Southern University of Science and Technology. He received his B.E. degree in computer science and technology from Southern University of Science and Technology in 2021. His research interests are mainly in blockchain, network economics, and consensus protocols.

Qingsong Wei received the PhD degree in computer science from the University of Electronic Science and Technologies of China, in 2004. He was with Tongji University as an assistant professor from 2004 to 2005. He is a Group Manager and principal scientist at the Institute of High Performance Computing, A*STAR, Singapore. His research interests include decentralized computing, Blockchain and federated learning. He is a senior member of the IEEE.

Siow Mong Rick Goh received his Ph.D. degree in electrical and computer engineering from the National University of Singapore. He is the Director of the Computing and Intelligence (CI) Department, Institute of High Performance Computing, Agency for Science, Technology and Research, Singapore. His current research interests include artificial intelligence, high-performance computing, blockchain, and federated learning.