Hindawi
Security and Communication Networks
Volume 2023, Article ID 2413532, 14 pages
https://doi.org/10.1155/2023/2413532

Research Article

VM-Studio: A Universal Crosschain Smart Contract Verification and Execution Scheme

Tianxu Han,[1] **Jian Mao**,[1] **Sipeng Xie**,[1] **Qiyuan Gao**,[1] **Qin Wang**,[2] **Pinge Zhang**,[1] **and Yijia Fang**[1]

[1]*School of Cyber Science and Technology, Beihang University, Beijing 100191, China*
[2]*CSIRO Data61, Sydney, Australia*

Correspondence should be addressed to Jian Mao; maojian@buaa.edu.cn

Received 11 August 2022; Revised 29 October 2022; Accepted 22 February 2023; Published 18 April 2023

Academic Editor: Robert H. Deng

Blockchain interoperability promotes value delivery, application expansion, and ecological compatibility across heterogeneous blockchain systems. However, the contract framework and virtual machine construction in these systems are significantly different, and crosschaining becomes a challenging issue for system universality and compatibility. Starting from this problem, in this study, we propose VM-Studio, a crosschain smart contract verification and execution scheme to migrate the virtual machines (VMs) from the origin blockchain to the target blockchain. In our scheme, the migrated VMs are loaded as independent components enclosed in containers. We also design a unified system schedule to enable VM-Studio to allocate transactions into different containers. Loaded with origin blockchain VMs, these containers can accordingly solve crosschain transaction execution and smart contract verification. We implement VM-Studio and evaluate the transaction execution performance in the origin environment with multiple blockchains and the container environment. Experiment results demonstrate that VM-Studio achieves broad universality without compromising the execution performance of original blockchain transactions.

1. Introduction

A major benefit provided by blockchain is the ability to connect isolated individuals in a direct peer-to-peer way. Since Nakamoto [1] proposed Bitcoin in 2008, blockchain technology has been developed for over a decade, and various blockchain systems emerge constantly innovating in many aspects, such as consensus security [2, 3], privacy protection [4, 5], smart contract computing [6, 7], and blockchain scalability [8, 9]. However, this has also led to the phenomenon that they have formed separate value systems. The information barrier across heterogeneous blockchain systems makes value circulation and information transmission a complex problem. Therefore, crosschain technology, also known as interoperability [10], is equally important as a fundamental property to establishing Web3 ecosystems.

Crosschain technology is mainly oriented towards two aspects of the issue between blockchains, one is value exchange and the other is data transfer [11]. In the early understanding of crosschain technology, people only cared about how their assets were transferred from the one chain to another [12]. With the advent of the era from blockchains are smart contracts, assets no longer exist in the form of pure native tokens in the blockchain transaction sheet and tokens defined and managed by smart contracts gradually occupy the mainstream market [13]. In addition, in various smart contract ecosystems, algorithms [14] and components with powerful functions and data precipitation abound, making many blockchains have unique crosschain value. In order to make the crosschain value and mean of blockchain widespread, for a specific blockchain, we call it original blockchain. In the crosschain scenario, we will consider all aspects: transfer the value in the transaction order, call the algorithm controlled by the smart contract, and even migrate the upper Dapp application.

We note that the core of the above crosschain services is to verify the correctness of transaction orders and smart

contracts on other blockchains on the chain. It is not very easy to achieve between heterogeneous blockchains with different consensus mechanisms, single constructs of transactions, underlying contracts, and data interfaces. The existing crosschain projects need to negotiate with the target chain, including agreement, consensus, a trusted middleman, side-chain return to the target chain confirmation, and other operations. These schemes have a common problem: they are limited by the contract of the target chain and the architecture of the underlying virtual machine. When the contract language of the target chain is changed or the virtual machine of the target chain is upgraded, the corresponding crosschain scheme is no longer available. Therefore, a universal and multichain compatible universal scheme is needed for crosschain operation.

Considering the following scenario, two blockchains, the original blockchain and the target blockchain, are denoted as C_O and C_T, respectively. The state data controlled by the contract on C_O are transported to C_T, which is used for the construction of crosschain transactions by C_T. This process can be easily achieved if C_T supports smart contracts and is compatible with C_O's virtual machine (VM). As a result, crosschain smart contract verification can be easily realized by inputting the relevant transaction order and smart contracts from C_T into C_T's VM. However, the differences in consensus mechanisms, contract construction, and virtual machine construction between two blockchains become a huge hindrance. It is not easy to achieve interoperability due to their heterogeneity. If the underlying logic code of C_T's VM is modified or the data structures on C_O are modified to be compatible with the C_T's VM, the problem can be greatly mitigated. Moreover, it is also difficult for C_T to access the crosschain system for other heterogeneous blockchains. It can be seen that the construction of crosschain verification and execution schemes for smart contracts requires rigorous requirements on VM construction for both parties.

To sum up, after the abovementioned analysis, we give the primary motivation of the proposed scheme here. We note that crosschain technology mainly involves three parts: the transfer of value in the transaction order, the crosschain invocation of smart contracts, and the transplantation of Dapp ecology. All of them are related to the underlying verification logic of the blockchain virtual machine. We were inspired by Nervos CKB Polyjuice [15], which ran an EVM instance using CKB VM to implement a blockchain running a native account model within a blockchain of UTXO models. We believe that if this part of executing transactions and verifying smart contracts of blockchain virtual machines is decoupled separately and encapsulated by accessible unified services, crosschain execution and verification of transactions and contracts on heterogeneous blockchains can be realised, and it has the characteristics of strong universality, is easy to deploy and start, and is easy to upgrade.

1.1. Our Solution. We propose VM-Studio, a universal crosschain smart contract verification and execution scheme. We address the interoperability problems across heterogeneous blockchains by transforming the verification and execution of smart contracts into virtual machine migrations. Supposing to verify the contract states of C_O on C_T, we load the virtual machine image of blockchain C_O into a bare container. In an execution environment of C_T, C_O's transaction order is imported into the container as an atomic task. Then, C_O's VM execution environment can be simulated to verify and execute relevant transaction orders and smart contracts. VM-Studio allows any blockchain system to import its VM image into an empty container of another VM-Studio component (heterogeneous blockchain node), making the execution environment a *container service + virtual machine image*. Then, VM-Studio can manage and invoke all types of containers through scheduling tools. Due to the characteristics of container service, our scheme has no additional requirements on the VM construction among heterogeneous blockchains, showing strong universality. Based on that, *our contributions* are summarised as follows:

(i) We propose VM-Studio, a universal crosschain smart contract verification and execution scheme. We give the concrete construction of VM-Studio and elaborate on the message execution processes and crosschain verification of smart contracts.

(ii) We analyze the advantages of the VM-Studio scheme, including its correctness, universality, and low overhead in the execution of origin transactions and smart contracts.

(iii) We conduct experiments on crosschain smart contracts' verification and execution performance overhead. We compare them with the running results of each origin blockchain. We conclude that VM-Studio has satisfactory performance outcomes.

1.2. Our Advantages. Compared to more common crosschain tools, our solution has the following advantages:

(i) VM-Studio is compatible with a wide range of heterogeneous blockchains. Compared with the more traditional crosschain schemes, such as [12, 16], our scheme faces all heterogeneous blockchains that support virtual machines and has no specific requirements on the single transaction structure and smart contract architecture of the blockchain itself, which is convenient for the unified transplantation of all kinds of blockchains.

(ii) The native blockchain environment is relatively secure, and its VM is easy to upgrade. Existing crosschain projects [17, 18] that use side chain, relay chain, or parallel chain architecture require additional data interfaces outside the chain for crosschain verification and face tricky data availability problems when the chain version is upgraded. Our solution will be native blockchain transactions and smart contracts executed in their encapsulated original environments to maximize the security of the transaction execution process.

(iii) The scheme has a simple structure and is easy to implement. Compared with some recent research

[19, 20], the blockchain is used as the crosslink mechanism, supplemented by a consensus mechanism to ensure security; alternatively, some solutions [19] achieve cross-chain interoperability by using a unified programming language to call the smart contracts on different chains; they have specific difficulties in the implementation; that is, the construction of crosschain environments puts forward higher requirements. Our solution decouples the functions of blockchain virtual machines and wraps the templates for verifying transactions and smart contracts separately into containers. In the simplest case, our crosschain verification can be implemented in the local environment of nodes.

1.3. Paper Structure. Section 1 describes the scientific problems, solutions, contributions, and advantages proposed in this study. Section 2 provides related work. Section 3 defines the parameters and assumptions. Section 4 introduces the main construction of the VM-Studio scheme and presents the detailed workflow of four types of VM-Studio messages. Section 5 gives our experimental design and corresponding results. Section 6 discusses the performance and application of VM-Studio from different perspectives. Last, Section 7 concludes this study.

2. Related Work

2.1. Smart Contract. A smart contract is essentially a piece of the program stored on the blockchain that can be automatically executed according to specified contract rules. The concept of a smart contract was first proposed by Szabo [21] in 1994, aiming to build an efficient contract without ambiguity that could be enforced with the help of code. Then, smart contracts were widely adopted with the advent of Ethereum [22]. In Ethereum, smart contracts are executed inside the Ethereum Virtual Machine (EVM) and are isolated from external environments. To ensure the reasonable usage of resources, Ethereum adopts the gas billing standard, where any computing operation needs to pay a certain amount of gas as the cost, including the *creation*, *call*, *data acquisition,* and other contract operations. EoS [23] claims to propose a set of smart contract systems that are easier to develop, which solves the problems of low performance and high fees of Ethereum smart contracts. Hyperledger Fabric [24] is an open license blockchain designed for crossindustry development among enterprises. It is built on Linux with good modularity and structural characteristics [24]. It provides reliable services for many industries, as well as high portability and versatility. Chaincode, the smart contract on Hyperledger Fabric, runs in a Docker container. It is a program that can query or change the ledger's state. The final execution results are synchronized to every node in the network. However, due to the alliance chain's centralization concerns, Hyperledger Fabric is unsuitable for the public chain.

2.2. Blockchain Interoperability. Blockchain interoperability technologies focus on the verifiable transfer of on-chain tokens and states across different chains [11]. Typically, three types of methods have been adopted: *hash-time lock*, *third-party crosschain,* and *sidechains/relays* mechanisms.

2.2.1. Hash-Time Lock. The hash-time lock mechanism [16] adopts two core techniques: *hash lock* and *time lock*. Two parties first need to establish a communication channel. Then, the two locking techniques are used to lock their on-chain assets, and the negotiated hash value is used to unlock the on-chain holdings of the other party for crosschain exchanges. The hash-time lock has apparent disadvantages: both parties have to hold the corresponding accounts on two chains with sufficient assets, and both blockchains must support smart contracts for contract executions.

2.2.2. Third-Party Crosschain. The third-party crosschain mechanism, such as the notary mechanism [25], introduces a trusted individual or group as the notary by tracking the status of two chains, collecting evidence, and verifying the transaction to facilitate the crosschain trade. It can support the interoperability of heterogeneous blockchains. However, the solution can only support the crosschain functions for origin token exchanges, and both sides connected by the crosschain transactions have to bear significant centralization risks. Hash-time lock and third-party mechanisms are subject to architectural and security constraints [26].

2.2.3. Sidechains/Relays. The sidechains/relays crosschain mechanism [12, 17, 18, 27] solves the centralization risk. Assets on the main chain and sidechains are mapped one by one through the two-way anchoring technology. The lock and release mechanism realizes the bidirectional workflow between the main chain and side chains. The solution has advantages: it supports the invocation of a crosschain smart contract, which is no longer limited to simple token exchange [28]. However, it suffers the drawback of high costs because its adoption is limited. Among sidechains/relays schemes, the best known are Cosmos [17] and Polkadot [18]. Polkadot is a crosschain platform that combines heterogeneous chains. It realizes the network-wide diffusion of user anonymity and formal verification through parachains. In Polkadot, transactions on each parachain can be passed to other chains through the bridge, and these transactions can be executed and verified by multiple chains. However, Polkadot has problems such as difficulties in selecting a validator, governance, forming parachains, or intransparency of validator election [29]. Cosmos aims to build an interconnected blockchain ecosystem. It provides a relatively complete and convenient way of facilitating crosschain interaction through the *Tindermint* consensus engine, the *Cosmos SDK* modular development framework, and the interblockchain communication protocol. However, the system is relatively complex, which is difficult for the developers to understand, due to factors, such as its economic model and market competition, Cosmos suffers the pressure from nonpure technical reasons.

In addition, several studies on heterogeneous blockchain crosschain also deserve attention [19, 20]. HyperService [19] is a heterogeneous crosschain framework that programmers can freely develop. It provides a unified Dapp programming language at the bottom of the native blockchain and supports crosschain communication with smart contract construction. However, the development framework proposed by HyperService is relatively complex, such as the specific format of data, dynamical checks for accuracy, and data input interface requirements, which lack universality. The scheme proposed in this study has a lower migration cost than HyperService in terms of the new blockchain virtual machine because the former only requires simple VM image encapsulation and loading, while the latter needs to develop the underlying contract logic of the new blockchain into a unified interface, according to the relevant developer documents. When the smart contract or virtual machine version of the native blockchain needs to be upgraded, the HyperService must be redeveloped according to the new rules. Our solution only needs to call a virtual machine image of the latest version. Blockchain Router [20] establishes a set of crosslink routing rules for multiple blockchains to establish the routing path for crosschain communication. The process of crosschain data verification is completed by the nodes of the block link, which is different from the focus of this study.

3. Parameters and Assumptions

We denote a blockchain system by C. For multiple blockchain systems, use $C_1, \cdots, C_i, \cdots$, where i is a variable positive integer. For a blockchain system C_i, we make the following assumptions.

3.1. Chain Identifier. Every blockchain system, or chain, has a unique identity. The genesis block is identified with a string of hash value GenesisHash. For the blockchain C_i, we define its identifier id_{C_i} as follows:

$$id_{C_i} = \mathrm{H}_{C_i}\left(\mathrm{GenesisHash}_{C_i}\right). \tag{1}$$

Here, H_{C_i} is a hash function, where a random string of length $\{0,1\}^*$ is mapped to the range of id_{C_i}'s values. $\mathrm{GenesisHash}_{C_i}$ represents the hash value of the genesis block on C_i.

3.2. Chain State. By default, the state model of blockchain can be applied to both the *UTXO* and the *account*-based blockchains. For generality, we define the state model as follows.

Definition 1 (chain state). For the blockchain C_i, we define, when it is based on the account model, we denote the set of states corresponding to all accounts as the chain state of C_i; when it is based on the UTXO model, the set of states corresponding to all UTXOs is called the chain state of C_i.

It is worth noting that for the blockchain system with the account model, the chain state can be equivalent to the corresponding state root in the block header of the latest block, such as the concept of the world-state MPT root in Ethereum. For the blockchain system with the UTXO model, the chain state represents the set of outputs of all transactions for which the input from subsequent transactions has not been referenced: the set of all existing UTXOs on the blockchain. In either case, we abstract the chain state as a snapshot, which can reflect the latest state or data set of the blockchain in real time. Equivalently, there is a more straightforward method: execute all transactions on the blockchain in order, from the genesis block to the latest block. It can be seen that their essences are very similar.

Therefore, in this study, we mainly express the perspective of the account-based blockchain to have a more intuitive understanding. As described above, by specifying the chain state as a pointer field in the block header of the latest block, we can calculate and verify the states through the batch acquisition of transaction data. If all the transactions are known, the state database of the corresponding data structure can be calculated by existing rules. Then, the chain state pointer can be extracted by the corresponding rules. The correctness of the obtained block data can be verified by comparison. With the continuous generation of new caches, we can verify the correctness of new transaction orders by calling the cached snapshot. In addition, to facilitate the construction of invocation instances for crosschain smart contracts, we define a multichain system:

Definition 2 (multichain). A multichain system $M(n)$ is a multichain system consisting of a series of parallel blockchains $M = \{C_1, ..., C_n\}\,(n \geq 2)$ if for any $i \in [1, n]$, and for any $j \in [1, n]$, there is $id_{C_i} \neq id_{C_j}$.

The above feature ensures that a multichain system $M(n)$ does not contain two identical blockchains, even though their genesis blocks are consistent. Equivalently, when the Genesis blocks of two blockchains are the same, we allow the sets of states to be different, indicating that the two blockchains are branched from the same chain. Nevertheless, we do not care about this situation.

In addition, we introduce several concepts mentioned in the previous section. Consider the crosschain scenario of two blockchains, where we call the crosschain from the origin blockchain to the target blockchain, denoted by C_O and C_T, respectively.

4. Main Construction of VM-Studio

In this section, we begin with an overview of the main construction of VM-Studio, including its five main components and four message types. Later, in 4.2, we give the details of the four types of messages, the interaction process of the five main components of VM-Studio, the specific structure of the messages, and the verification process of the data.

4.1. Overview. The main construction of VM-Studio is shown in Figure 1. As a general chain state execution construction, VM-Studio components are attached to the

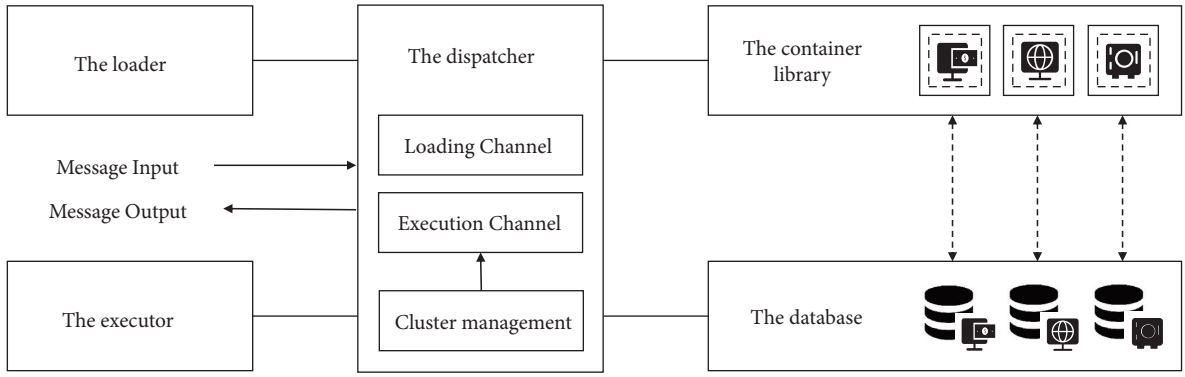

Figure 1: The structure of VM-Studio.

network nodes of a relay blockchain. The following sections will present concrete examples of running crosschain smart contracts under the intermediate blockchain. VM-Studio consists of five main modules: *the dispatcher*, which is responsible for overall dispatch, message input, and output; *the loader* is responsible for loading the VM image; *the executor*, which executes containers loaded with blockchain VMs; *the container library*, which stores the containers loaded with virtual machines; and *the database*, which is responsible for storing a snapshot of states as a database.

4.1.1. The Dispatcher. It is the core component of VM-Studio. It receives the input and output messages of message instructions externally and processes messages internally.

The dispatcher mainly receives four types of messages: *MessageI*, *Preload* mainly used to verify the correctness and integrity of packets and update the block synchronization height to start the subsequent formal loading process. *MessageII*, *load*, is mainly used to load the prepared virtual machine image into the empty container, transport it to the container library for storage, and open the access interface between it and the corresponding blockchain ledger in the database to start the subsequent execution process. *MessageIII*, *Execute* is mainly used to manage atomic transaction order messages through the cluster, and the container verifies transactions and executes smart contracts. *MessageIV*, *Preload and Update*, both the preload and virtual machine version update functions. The dispatcher first performs a preload process.

For the above four messages, the dispatcher has different message channels to process them: *MessageI*, *MessageII,* and *MessageIV* will be included in the loading channel; *MessageIII* is included in the execution channel and scheduled by the cluster management, an internal component of the dispatcher. This is because *MessageIII* is the atomic transaction order obtained after the block data are split, and the quantity is millions of times that of other types of messages. In order not to delay the processing of other types of messages but also to allocate the core resources for *MessageIII*, *MessageIII* is included in an independent message channel.

4.1.2. The Container Library. The container library is the container warehouse of VM-Studio, which is used to store containers after loading and presents the call interface with the blockchain *id* as the index. Cluster Management centrally arranges for the executor to call the container library.

4.1.3. The Database. The database is the back-end database of VM-Studio, which stores the blockchain ledger, state snapshot, state database, and virtual machine image. It presents the call interface with the blockchain *id* as the index. The VM can be directly called by the container where the VM of the corresponding blockchain resides.

4.1.4. The Loader. The loader is the loading center of VM-Studio. The loader is used to load virtual machine images into empty containers for *MessageII* services.

4.1.5. The Executor. It is the executor management center of VM-Studio. Triggered by *MessageIII*, the Cluster Management component of the dispatcher will send the call instruction to the executor for containers. The executor will then call the corresponding container in the container library through the call interface, that is, characterized by *id*.

Next, we will describe the workflow and interactions of VM-Studio after each type of message is given.

4.2. Details of the Scheme. In this section, we describe in detail how VM-Studio performs crosschain execution and verification of transactions and smart contracts when receiving the mentioned four types of messages, including those actively sent by nodes and those automatically generated by the dispatcher. The overall process is shown in Figure 2. Each type of message is a dataset packet dp consisting of a header dp_{head} and a body dp_{body}, while dp_{body} is mainly composed of block data.

4.2.1. Message Signature. First, we formally describe the user identity and signature of the node. In our setting, VM-Studio is positioned as a general blockchain crosschain data verification component; in a given blockchain system, a node that has installed and instantiated VM-Studio can verify

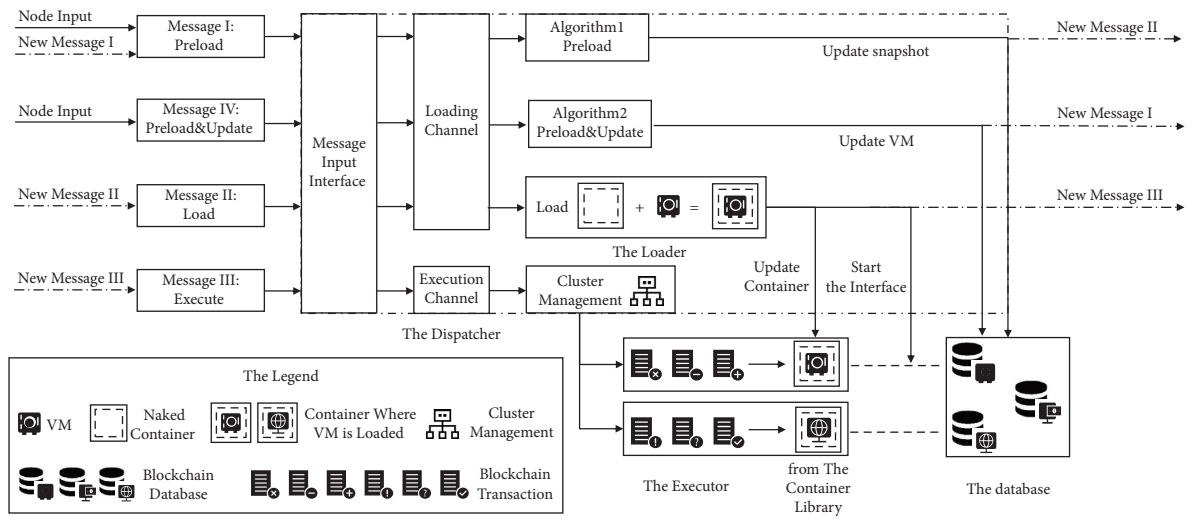

Figure 2: The operation flow of four kinds of messages.

crosschain data in the blockchain system. Considering that the identity of a node as a chain user is not fixed in different chain environments, we use a common representation of the public chain here. We have abstracted the more traditional blockchain mechanisms, such as Bitcoin and Ethereum. For a more detailed description, please refer to [1, 22]. For a node to sign the message M, the following three algorithms are formally defined, including the key generation algorithm KeyGen, the signature algorithm Sig, and the signature verification algorithm VerifySig.

$\mathrm{KeyGen}(\lambda, cp) \longrightarrow pk, sk, \mathrm{add\,r}$. λ is a security parameter, and cp is a blockchain parameter that contains information, such as the version byte of the blockchain used to distinguish between the main network and the test network. When a node joins the blockchain network, the private key sk is generated by a random number in a specific range and the public key pk is generated by sk in a trapdoor reversible way. Finally, the address $add\,r$ is generated by an irreversible algorithm: the address of the node's account in the blockchain network. It should be noted here that, first, the range of random numbers used to generate sk is determined by the security parameter λ. Second, we generally assume that the irreversible algorithm used is the hash function, H_{C_i}, and the calculation process is $add\,r = cp \| \mathrm{H}_{C_i}(pk)$.

$\mathrm{Sig}(M, sk) \longrightarrow sig_M$. sig_M is the result of signing the message M for a node using its own private key sk.

$\mathrm{VerifySig}(\mathrm{sig}_M, M, pk) \longrightarrow \{0, 1\}$. If the signature verification is successful, output is 1; otherwise, output is 0.

In general, in the practical application of blockchain, the message signature's object may be the message's hash value. In verifying the signature, it is necessary to verify that the related address $add\,r$ is from the specific chain version cp and matches the public key pk. In Algorithm 1, we present the verification procedure of the message signature when the input is $(sig, pk, \mathrm{add\,r}, cp, M^*)$.

4.2.2. MessageI, Preload. *MessageI* is initiated by the node or automatically initiated by the dispatcher after *MessageIV* is processed. In the previous article, we mentioned that *MessageI* is used to verify the correctness and integrity of data packets and update the block synchronization height to enable subsequent formal loading processes. When *MessageI* is correctly processed by the dispatcher, the dispatcher will automatically generate *MessageII* and send it to the input of the dispatcher's message channel. The specific process is as follows:

(i) *Step 1.* Input MessageI to the dispatcher.

(ii) *Step 2.* The dispatcher parses MessageI into

$$\left(\mathrm{type}, vpp_{\mathrm{sig}}, h_{\mathrm{start}}, h_{\mathrm{end}}, id\right). \tag{2}$$

The detailed structure of *MessageI* is listed in Table 1. According to $type$, the message type is *MessageI*.

(iii) *Step 3.* The dispatcher operates Algorithm 2, and accordingly outputs

$$\left(dp_{\mathrm{head}}, dp_{\mathrm{body}}, \mathrm{maxheight}, \mathrm{err}\right) \leftarrow \mathrm{PreLoad}\left(dp_{\mathrm{head}}, dp_{bpdy}\right). \tag{3}$$

If err equals to 1, break the process.

(iv) *Step 4.* The dispatcher parses dp_{head} to

$$\left(id, h_{\mathrm{start}}, h_{\mathrm{end}}, vpp_{\mathrm{sig}}\right), \tag{4}$$

and sends

$$\left(dp_{\mathrm{body}}, h_{\mathrm{start}}, h_{\mathrm{end}}, id\right), \tag{5}$$

as *MessageII* to the dispatcher's message channel.

4.2.3. MessageII, Load. *MessageII, Load* is automatically initiated by the dispatcher after MessageI is processed. Its main functions include loading the prepared virtual machine image into an empty container, transporting it to the container library for storage, and opening the access interface between it and the corresponding blockchain database for subsequent executions. When *MessageII* is correctly

Input: sig, pk, add r, cp, M^*
Output: b
Initialization: 1⟵b
If VerifySig(sig, $H_{C_i}(M^*)$, pk)! = 1 **then**
  0⟵b;
  return b;
end
If $add\,r$! = $cp \| H_{C_i}(pk)$ **then**
  0⟵b;
  return b;
end
return b

Algorithm 1: VerifyMessageSig.

Table 1: Message structure of *MessageI*.

Parameters	Meanings
type	Message type and it has the value I
vpp_{sig}	Publicly verifiable signature parameters, including parameters related to node identity: (i) sig: the node's signature for hash_{dp} (ii) pk: the node's public key (iii) add r: the node's address in blockchain network (iv) cp: the blockchain parameter that contains information such as the version byte of the blockchain used to distinguish between the main network and the test network (v) hash_{dp}: the hash value of the data package
h_{start}	The starting height of the synchronization block
h_{end}	The ending height of the synchronization block
id	The identifier of the blockchain from which the packet originated

Input: dp_{head}, dp_{body}
Output: dp_{head}, dp_{body}, maxheight, err
Initialization: 0⟵b, 0⟵err, 0⟵maxheight
parse $(id, h_{\text{start}}, h_{\text{end}}, vpp_{\text{sig}})$⟵$dp_{\text{head}}$
parse $(\text{sig}, pk, \text{add r}, cp, \text{hash}_{dp})$⟵$vpp_{\text{sig}}$
parse GenesisHash ∈ dp_{body}
run b⟵VerifyMessageSig(sig, pk, add r, cp, hash_{dp}), $b \in \{0, 1\}$
If b == 0 **then**
  1⟵err;
  return err;
end
If H(GenesisHash)! = id **then**
  1⟵err;
  return err;
end
query sn and maxheight for id
If h_{start} > maxheight or maxheight ≥ h_{end} **then**
  1⟵err;
  return err;
else
  h_{start}⟵maxheight;
  maxheight⟵h_{end};
end
package dp_{head}⟵$(id, h_{\text{start}}, h_{\text{end}}, vpp_{\text{sig}})$
return dp_{head}, dp_{body}, maxheight, err

Algorithm 2: Preload.

processed by the dispatcher, the dispatcher will automatically generate *MessageIII* and send it to the input of the dispatcher's message channel. The main process is as follows:

(i) *Step 1*. Input *MessageII* to the dispatcher.

(ii) *Step 2*. The dispatcher parses *MessageII* to

$$\left(\text{type}, dp_{\text{body}}, h_{\text{start}}, h_{\text{end}}, id\right). \tag{6}$$

The detailed structure of *MessageII* is listed in Table 2. According to type, the message type is *MessageII*.

(iv) *Step 3*. The dispatcher loads the blockchain VM corresponding to blockchain *id* and inputs it to the loader. Meanwhile, the database interface corresponding to blockchain *id* is opened to the loader.

(v) *Step 4*. The loader uses the bare container to load the corresponding VM, which is expressed as Container$(VM(id))$.

(vi) *Step 5*. The loader links Container$(VM(id))$ to the interface of blockchain *id* in the database.

(vii) *Step 6*. The loader starts Container$(VM(id))$ and synchronizes blockchain *id* ledger from h_{start} to h_{end} to load data.

(viii) *Step 7*. After data are loaded, the dispatcher parses dp_{body} to atomic transactions and transmits each transaction to the input of the dispatcher's message channel in the form of *MessageIII*.

4.2.4. MessageIII, Execute. MessageIII, Execute is automatically initiated by the dispatcher after MessageII is processed. Its main use is to distribute atomic transaction in single messages through Cluster Management to the corresponding container, which then, guided by the executor, validates the transaction and executes the smart contract.

(i) *Step 1*. Input *MessageIII* to the dispatcher.

(ii) *Step 2*. The dispatcher parses *MessageIII* into

$$(\text{type}, id, tx). \tag{7}$$

The detailed structure of *MessageIII* is listed in Table 3. According to $type$, it is *MessageIII*. The dispatcher allocates this message to the execution channel.

(iii) *Step 3*. The executor passes the corresponding tx to the working Container$(VM(id))$ according to the *id* information in Message. If the container is not in working state, the executor fetches Container$(VM(id))$ from thecontainerlibrary according to *id*.

4.2.5. MessageIV, Preload and Update. MessageIV, Preload and Update is initiated by the node itself and can be regarded as an extended version of *MessageI*. It can preload and update virtual machine versions. The dispatcher will first perform a preload process and then determine whether the virtual machine of the corresponding blockchain needs to be updated. If so, the virtual machine image stored in the database will be updated. After the update, the dispatcher will automatically generate *MessageI* and moreover feed it to the input part of the dispatcher's message channel. The main process is as follows:

(i) *Step 1*. Input *MessageIV* to the dispatcher.

(ii) *Step 2*. The dispatcher parses *MessageI* into

$$\left(\text{type}, vpp_{\text{sig}}, h_{\text{start}}, h_{\text{end}}, id, VM - \text{version}\right). \tag{8}$$

The detailed structure of *MessageIV* is listed in Table 4. According to $type$, the message type is *MessageIV*.

(iii) *Step 3*. The dispatcher operates Algorithm 3 and then outputs

$$\left(dp_{\text{head}}, dp_{\text{body}}, \text{maxheight}, \text{isupdate}, \text{err}\right) \leftarrow \text{PreLoad\&Update}\left(dp_{\text{head}}, dp_{bpdy}\right). \tag{9}$$

If *err* equals to 1, break the process.

(iv) *Step 4*. If isupda te equals to 1, dispatcher parses dp_{head} into

$$\left(id, h_{\text{start}}, h_{\text{end}}, vpp_{\text{sig}}, VM - \text{version}\right). \tag{10}$$

The dispatcher updates the virtual machine corresponding to blockchain *id* of the database to the $VM - version$ version. The dispatcher packages

$$dp_{\text{head}} \leftarrow \left(id, h_{\text{start}}, h_{\text{end}}, vpp_{\text{sig}}\right), \tag{11}$$

and sends $(dp_{\text{head}}, dp_{\text{body}})$ as *MessageI* to the input of the dispatcher's message channel.

(v) *Step 5*. If isupda te does not equal to 1, dispatcher will parse

dp_{head} to $(id, h_{\text{start}}, h_{\text{end}}, vpp_{\text{sig}}, VM - \text{version})$ and sends

$$\left(dp_{\text{body}}, h_{\text{start}}, h_{\text{end}}, id\right), \tag{12}$$

as *MessageII* to the dispatcher's message channel.

5. Implementation and Experiment

In this section, we implement the VM-Studio prototype and report our experiment results by comparing our prototype with the original blockchain node implementations. We

Table 2: Message structure of *MessageII*.

Parameters	Meanings
type	Message type and it has the value *II*
dp_{body}	The body of the data packet, which contains the block data of the blockchain
h_{start}	The starting height of the synchronization block, included in the outputs of *MessageI*
h_{end}	The ending height of the synchronization block, included in the outputs of *MessageI*
id	The identifier of the blockchain from which the packet originated, which equals to that in *MessageI*

Table 3: Message structure of *MessageIII*.

Parameters	Meanings
type	Message type and it has the value *III*
id	The identifier of the blockchain from which the packet originated, which equals to that in *MessageII*
tx	An indivisible atomic transaction

Table 4: Message structure of *MessageIV*.

Parameters	Meanings
type	Message type and it has the value *IV*
vpp_{sig}	Publicly verifiable signature parameters, including parameters related to node identity: (i) sig: the node's signature for hash_{dp} (ii) *pk*: the node's public key (iii) add r: the node's address in blockchain network (iv) *cp*: the blockchain parameter that contains information such as the version byte of the blockchain used to distinguish between the main network and the test network (v) $hash_{dp}$: the hash value of the data package
h_{start}	The starting height of the synchronization block
h_{end}	The ending height of the synchronization block
id	The identifier of the blockchain from which the packet originated
VM – version	The VM version of the blockchain from which the packet originated

implemented our prototype in three unstable, dynamic environments and applied it to than 1,300,000 blocks with 1,000,000 transactions for testing.

5.1. Implementation Details. We use an elastic compute server with a 2.1 GHz Intel Xeon(R) gold 6,230 CPU, a 32 GB of 2,400 MHz DDR4 memory, a 1 TB solid state disc, and an Ubuntu 18.04 operating system to implement the proposed system. The node implementation (*Go-Ethereum*) was modified and could interact with the premade containers, which loaded virtual machines from each involved blockchain; we also employed the *go-metrics* package to add metrics to measure the execution time of the *contract call* function, the *contract create* function, and the RPC communication function. Full nodes are running on a proof-of-work blockchain named *Ropsten*, while operating on a proof-of-authority blockchain named *Görli* and the blockchain with a hybrid consensus engine named *BnB Smart Chain* (*BSC*, for short). For each blockchain, we test our prototype with more than 1,300,000 blocks and up to 1,000,000 transactions. Then, we compare the overhead of the origin VM with our implementation. The results are as follows.

5.2. Experiment Results. As described above, we run experiments on three blockchain networks: *Ropsten*, *Görli*, and *BnB Smart Chain*. To test out the performance of our prototype system, we measure the overhead of both the RPC communication and the virtual machine computation; after that, we conduct an overall overhead comparison between the VM-Studio and the origin system. The full node of each blockchain is deployed on the elastic compute server mentioned previously.

Firstly, we discuss the virtual machine computation overhead of VM-Studio. The three blockchain networks contain different transaction sets, resulting in different computation performances. As shown in Figure 3, the average computation overhead of the *Ropsten* blockchain goes up during the first 10,000 transactions and reaches the highest point of around 285 microseconds and then goes slightly down to about 150 microseconds until the total of 1,000,000 transactions are applied. Comparing the performance of our prototype system with the original system, the latency our system adds on is less than 35 microseconds, which may be a result of the data exchange between different functions inside the container. From our view, the added latency is acceptable for production usage.

Input: dp_{head}, dp_{body}
Output: dp_{head}, dp_{body}, $maxheight$, err
Initialization: 0←b, 0←err, 0←$maxheight$ parse $(dp_{head}, VM - version)$←$dp_{head}$
run Algorithm 2: $(dp_{head}, dp_{body}, naxheight, err)$←Preload (dp_{head}, dp_{body})
query VM version $version$ for id
If $VM - version > version$ **then**
  1←$isupdate$;
end
package dp_{head}←$(id, h_{start}, h_{end}, vpp_{sig}, VM - version)$
return $dp_{head}, dp_{body}, maxheight, isupdate, err$

Algorithm 3: Preload and Update.

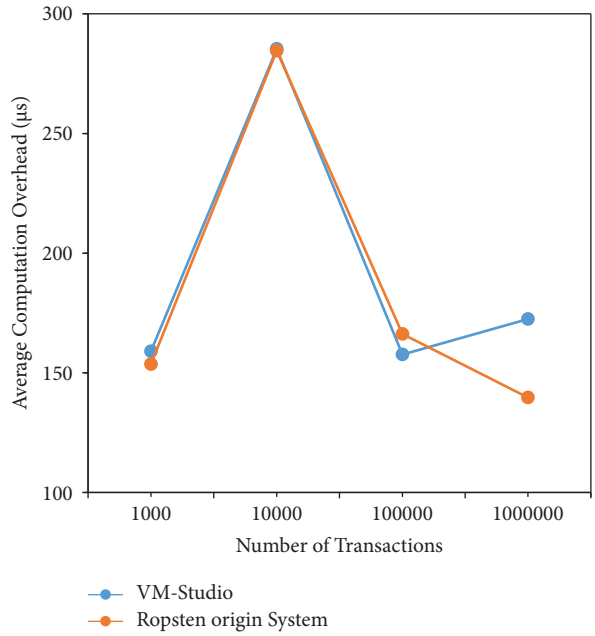

Figure 3: The average computation overhead on the Ropsten blockchain. The sampling size is fixed at 1,000,000 transactions, involving 402,361 blocks.

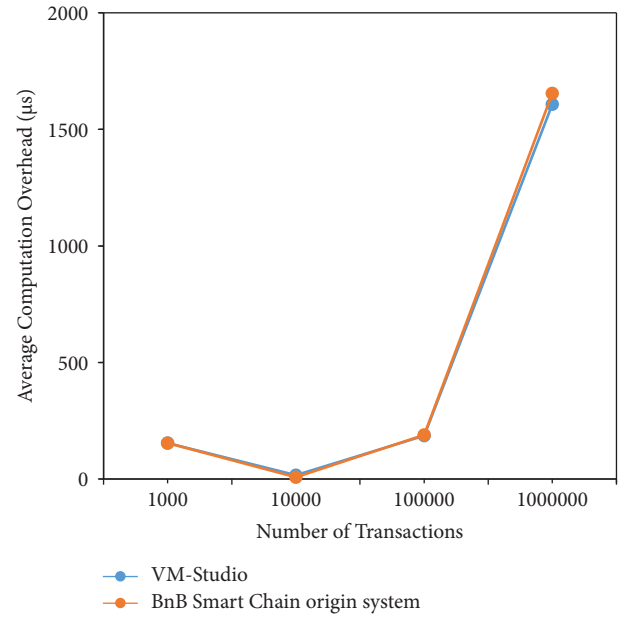

Figure 4: The average computation overhead on *BnB SmartChain*. The sampling size is fixed at 1,000,000 transactions, involving 1,311,838 blocks.

As for the *BnB smart contract* blockchain network, referring to Figure 4, the average computation overhead continues going down to about 10 microseconds within the first 10,000 transactions. Then, it keeps rising until it hits the summit of more than 1,600 microseconds when the virtual machine goes through the entire 1,000,000 transactions. The performance of both systems is relatively close, and our system performs slightly better when 1,000, 100,000, and 1,000,000 transactions come in. The reason is that transactions in the *BnB SmartChain* blockchain network are often related to more sophisticated smart contract codes. Therefore, our efficient EVM implementation presents better performance. VM-Studio offers about 8% performance improvement (computed by $(VO - OO)/OO$, where VO stands for VM-Studio overhead and OO stands for origin overhead) though it is on a microsecond scale.

Figure 5 shows the average computation overhead on the *Görli* blockchain network. The overhead fluctuates. It hits the summit of about 500 microseconds when the virtual machine executes about 10,000 transactions and comes down to 270 microseconds when 100,000 transactions come in. The performance difference is pretty apparent, and the statistics indicate that the highest performance gap between the two systems is around 40 microseconds when 10,000 transactions are executed.

Besides the computation overhead of the system, we also investigate the RPC communication latency added by VM-Studio. We use *gRPC* to transfer the RLP-encoded [22] transaction data between the Geth client and the virtual machine container. Figure 6 describes the average communication overhead of VM-Studio, which is caused by the data exchange between the *Geth* client and the virtual machine container. The communication overheads that VM-Studio adds when running on the Görli and Ropsten show downward trends. They start at about 2,000 microseconds when 1,000 transactions are executed and finally go down to about 400 microseconds. However, unlike the previous results, the overhead on the *BSC* stayed stable at about 300 microseconds during the whole

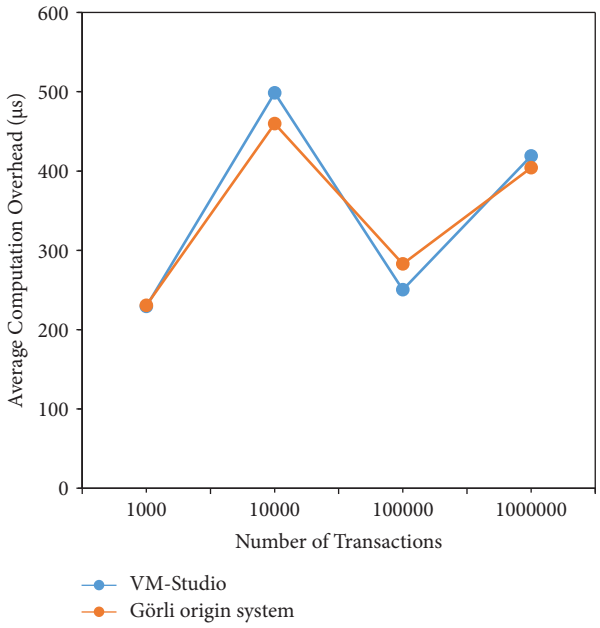

Figure 5: The average computation overhead on the *Görli* blockchain. The sampling size is fixed at 1,000,000 transactions, involving 1,806,736 blocks.

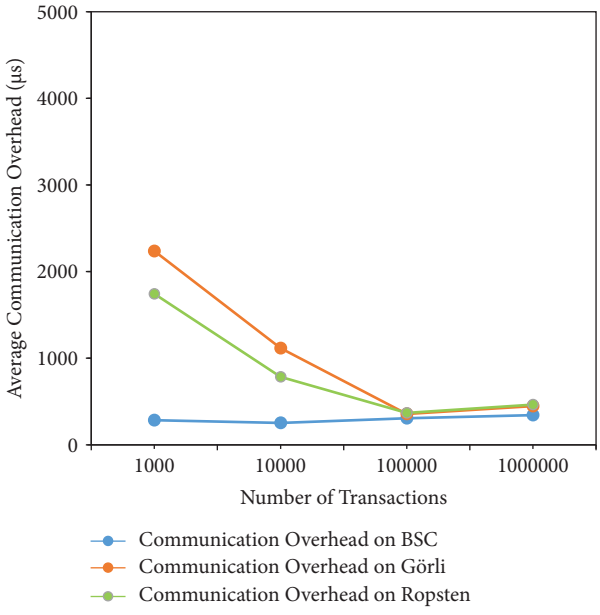

Figure 6: Extra communication overhead of VM-studio on three different blockchain networks.

procedure. The differences may mainly result from the characteristics of different blockchain systems, as the *Görli* and *Ropsten* are test networks that might contain some large transactions when they are first launched. While the *BSC* is the main network, its users will be more likely to consider the size of each transaction.

Finally, we put the computation overhead and the RPC communication overhead together to inspect the overall overhead of the VM-Studio. As we can see from Figure 7, the latency VM-Studio adds on continuously goes down with

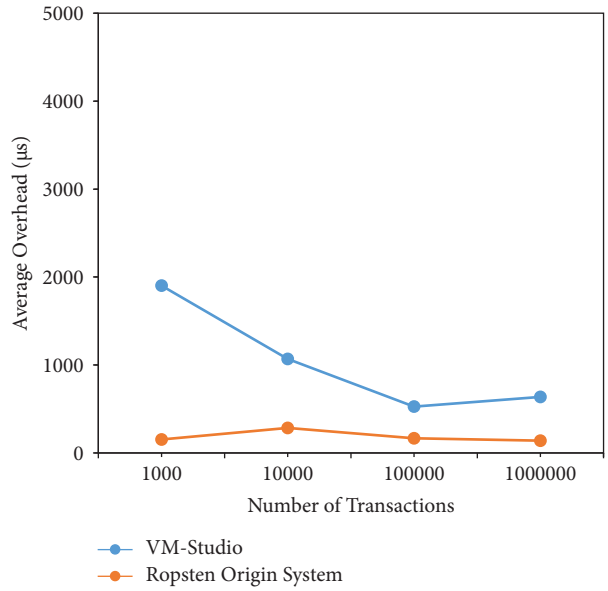

Figure 7: The average overhead on the *Ropsten* blockchain. The sampling size is fixed at 1,000,000 transactions, involving 402,361 blocks.

the increment of the transactions. The extra overhead stays about 450 microseconds when the 1,000,000 transactions are executed, mainly due to the RPC communication overhead that the VM-Studio adds. Furthermore, the situation on *Görli* is quite similar, which we can read in Figure 8, and the communication overhead is the main factor that affects the performance of the VM-Studio. As for the *BSC*, Figure 9 indicates that the performance of VM-Studio is quite close to that of the origin system. With about 300 extra microseconds, the difference between the performances of both systems is stable.

In conclusion, our experiments show that VM-Studio achieves availability. Due to the size of the transactions and the smart contract codes related to them, the prototype system's performance varies. As for communication costs, VM-Studio may add up to about 2000 microseconds to the origin system, mainly due to large transactions. Moreover, when economic factors restrict the sizes of transactions in the real-world production environment, the communication overhead is about 400 microseconds. Besides, the computation cost of the VM-Studio fluctuates around that of the original system. The maximal latency the VM-Studio adds is less than 40 microseconds, which is less significant compared with the communication overhead. Above all, the overhead added by the VM-Studio is majorly influenced by the communication overhead, which is about 400 microseconds in the real-world production environment. Therefore, we conclude that the extra latency is acceptable.

6. Discussion

6.1. Analysis of VM-Studio Scheme. The primary goal of VM-Studio is to ensure correct executions of both origin blockchain transactions and smart contracts. Here, as we

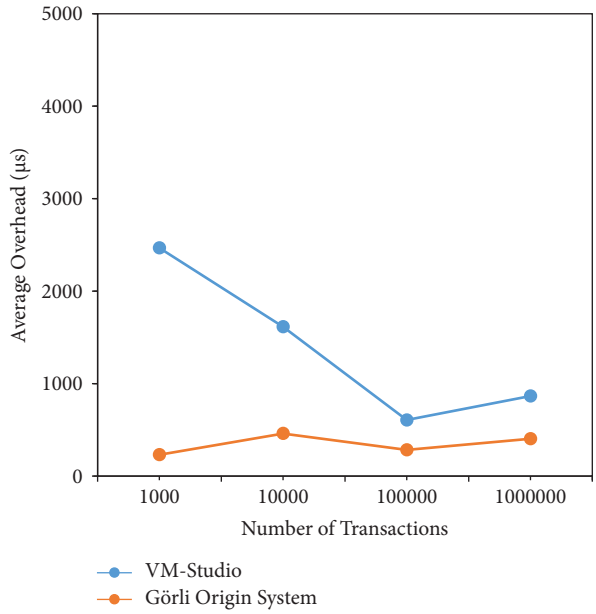

Figure 8: The average overhead on the *Görli* blockchain. The sampling size is fixed at 1,000,000 transactions, involving 1,806,736 blocks.

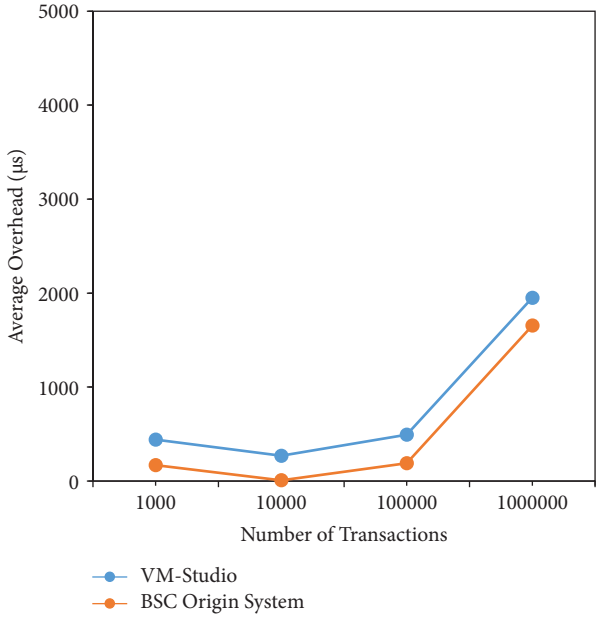

Figure 9: The average overhead on *BnB SmartChain*. The sampling size is fixed at 1,000,000 transactions, involving 1,311,838 blocks.

know, the execution of smart contracts is based on transactions. Therefore, we only discuss the correctness of transaction executions in the origin blockchain under our VM-Studio architecture.

We considered two factors in the architecture design of VM-Studio. First, the numbers and types of origin blockchains are various. VM-Studio is required to manage the resources of corresponding blockchains in a unified manner. We schedule the packaged containers into the container library for unified management to facilitate the overall migration or sharing of VM-Studio by nodes. Virtual machine re-encapsulation is not required; instead, the system offers unified interfaces to facilitate frequent fetching by the executor. In the database, containers provide the following unified interfaces: VM images, facilitating the first container loading and subsequent VM version checking and upgrading. Also, state snapshots are used by a VM to load data from the corresponding states and state database to import data into internal virtual machine images. Second, the origin blockchain transaction data are quite large, and each transaction will be sent to the container for execution as an atomic transaction. Therefore, the amount of transaction data multiplied by the number of origin blockchains will be millions. To solve the problem, we separate incoming messages of such atomic transactions from single-digit messages and place them in two different message queues. Thus, the cluster management software can allocate a large number of resources to the system for centralized processing of transactions without delaying the processing of other messages.

In addition, we have ensured this in the specific process design for four types of messages. First, when the message is input to VM-Studio for the first time, the system authenticates the packet header of the message through Algorithm 2. On the one hand, it verifies the signature of the message source. On the other hand, it dynamically adjusts the synchronization of the related origin blockchain in VM-Studio to facilitate subsequent loading processes. Second, VM images are loaded into containers from the origin blockchain. Transactions according to the original order have been continuously input into the container. The container's internal execution environment and that of the origin blockchain are entirely consistent, as long as the origin blockchain consensus has no objections to transaction results, which can ensure that transactions trading in VM-Studio perform correctly. Third, updating the virtual machine version will affect the transaction execution results. Since the update frequency of the virtual machine version is not high, we provide *MessageIV* to realize the update of the virtual machine in the database by VM-Studio.

6.2. Universality and Overhead. We explain that the VM-Studio solution is universal for the origin and target blockchains. Here, we illustrate two aspects. First, for an origin blockchain, the virtual machine means a machine that can automatically execute specific formats and certain types of transactions, and the relevant execution rules and verification rules have been hard coded inside the virtual machine. We load the virtual machine image into the container; thus, the container contains all the virtual machine rules, presenting corresponding data interfaces to the outside environment. Second, VM-Studio can be regarded as a set of components, which have little dependency on the blockchain architecture, and can be deployed at any node of the target blockchain. Therefore, VM-Studio also shows universality for the target blockchain. In addition, regarding the performance of VM-Studio to execute transactions on the target blockchain, in Section 6, experimental results have

shown that executing transactions on VM-Studio is slightly less efficient than those on the original system. The reason is that the transaction execution time related to sophisticated contract codes is mainly affected by the virtual machine, and the communication complexity inside and outside the container is insignificant in front of the computational complexity of the established transaction execution program of the virtual machine. However, the communication complexity mentioned above will take the lead when confronting simple transactions.

6.3. Read and Write Ability. We try to give a crosschain smart contract invocation example based on VM-Studio. Suppose that there exists a multichain system $M(n) = \{C_1, ..., C_{n-1}, C_n\}$, where $C(n)$ is a blockchain dedicated to initiating crosschain smart contract calls with VM-Studio components deployed on its nodes. While the VM images of $\{C_1, ..., C_{n-1}\}$ have been loaded into the VM-Studio container and their blockchain ledger, VM (latest version) images and state snapshots are stored in the VM-Studio database. At this point, we can consider that the blockchain C_n can run smart contracts on other chains.

As we know, calling a smart contract can be abstracted into two basic instructions: *Read and Write*. For a general blockchain system C, the usage of *Read* instruction only reads the chain state S_C but does not cause the change of S_C. Therefore, based on the world state of blockchain C and the execution environment of virtual machines, the *Read* instruction does not need to participate in the consensus of C to complete. However, the *Write* instruction directly changes the chain state S_C of the blockchain C. This process requires the consensus of chain C. Therefore, if a crosschain smart contract call transaction contains many *Write* instructions for different blockchain states, the system where VM-Studio is located is difficult to achieve. In particular, we specify that *Write* directives also include *Read* directives.

Now, consider a simple case where there is at most one crosschain smart contract call to a *Write* instruction. We give the following example:

(i) *Step 1.* Construct a crosschain smart contract call transaction, denoted as follows:

$$\left(id = id_{C_n}, \text{Read}\left(C_1, C_2, \cdots, C_i\right), \text{Write}\left(C(n)\right)\right). \quad (13)$$

The target chain of this transaction is $C(n)$, including chains $\{C_1, C_2, ...\}$, the *Read* instruction on C_I, and the *Write* instruction on the chain $C(n)$.

(ii) *Step 2.* Submit this transaction to VM-Studio, thus dividing it into $i + 1$ atomic transaction:

$$\text{Read}\left(C_1\right), \cdots, \text{Read}\left(C_i\right), \text{Write}\left(C_n\right). \quad (14)$$

(iii) *Step 3.* The above $i + 1$ atom transactions are presented to cluster management sequentially, and the corresponding container is further invoked through the executor to execute previous i atom transactions.

(iv) *Step 4.* Call $\text{Container}\left(VM\left(id_{C(n)}\right)\right)$ to perform $\text{Write}\left(C_n\right)$.

(v) *Step 5.* Trade execution results and submit

$$\left(id = id_{C_n}, \text{Read}\left(C_1, C_2, \cdots, C_i\right), \text{Write}\left(C(n)\right)\right), \quad (15)$$

to the blockchain.

So far, we have achieved a simple single-write crosschain smart contract call transaction by VM-Studio in the heterogeneous chain environment. In the above transaction

$$\left(id = id_{C_i}, \text{Read}\left(C_1, C_2, \cdots, C_i\right), \text{Write}\left(C(n)\right)\right), \quad (16)$$

the transaction should be submitted to the blockchain C_i for confirmation after the consensus. However, this problem can be addressed if a VM-Studio component is used on blockchain C_i.

The invocation scheme of the crosschain smart contract with a multiwrite type needs to be realized by the locking mechanism and incentive mechanism under the premise of VM-Studio, starting from the atomicity of crosschain transactions. We will focus on this issue in the future.

7. Conclusion

The heterogeneity of blockchain is one of the significant factors hindering crosschain schemes. This study proposes VM-Studio, a universal crosschain smart contract verification and execution scheme. The main idea of VM-Studio design is to transform the compatibility and adaptation of the original transaction execution construction, namely, virtual machine construction, into the migration and encapsulation of origin blockchain virtual machines. By establishing a close virtual machine container and providing a unified data interface, the transaction execution environment of all VM-supported origin blockchains can be simulated on the target blockchain to complete the verification of crosschain smart contracts. Through theoretical analysis and experimental verification, we conclude that VM-Studio has negligible performance loss compared with the origin blockchain when executing transaction orders within the order of 100,000. Finally, we give an example of a single-write invocation towards crosschain smart contracts to demonstrate the feasibility and applicability of VM-Studio.

Data Availability

The data used to support the finding of this study are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This study was supported by the National Key R&D Program of China through project 2020YFB1005600, the Natural Science Foundation of China through projects U21A20467,

61932011, and 61972019, the Beijing Natural Science Foundation through project M21031, and the Populus Euphratica Found CCF-Huawei BC2021009.

References

[1] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system," *Decentralized business review*, vol. 21260, 2008.

[2] A. Kiayias, A. Russell, and B. David, "Ouroboros: a provably secure proof-of-stake blockchain protocol," in *Proceedings of the Annual International Cryptology Conference*, pp. 357–388, Santa Barbara, CA, USA, August 2017.

[3] R. Pass and E. Shi, "Thunderella: blockchains with optimistic instant confirmation," in *Proceedings of the Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 3–33, Tel Aviv, Israel, April 2018.

[4] M. Campanelli, R. Gennaro, and S. Goldfeder, "Zero-knowledge contingent payments revisited: attacks and payments for services," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 229–243, New York, NY, USA, October 2017.

[5] Q. Wang, B. Qin, J. Hu, and F. Xiao, "Preserving transaction privacy in bitcoin," *Future Generation Computer Systems*, vol. 107, pp. 793–804, 2020.

[6] C. Schneidewind, I. Grishchenko, and M. Scherer, "Ethor: practical and provably sound static analysis of ethereum smart contracts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pp. 621–640, Virtual Event, USA, November 2020.

[7] M. Wang and Q. Wu, "Lever: breaking the shackles of scalable on-chain validation," 2019, https://eprint.iacr.org/2019/1172.%202019.

[8] C. Li, P. Li, and D. Zhou, "A decentralized blockchain with high throughput and fast confirmation," in *Proceedings of the 2020 {USENIX} Annual Technical Conference (USENIX ATC)*, pp. 515–528, Boston, MA, USA, July 2020.

[9] Q. Wang and R. Li, "A weak consensus algorithm and its application to high-performance blockchain," in *Proceedings of the IEEE INFOCOM 2021-IEEE Conference on Computer Communications (INFOCOM)*, pp. 1–10, Vancouver, BC, Canada, May 2021.

[10] R. Belchior, A. Vasconcelos, S. Guerreiro, and M. Correia, "A survey on blockchain interoperability: past, present, and future trends," *ACM Computing Surveys*, vol. 54, no. 8, pp. 1–41, 2021.

[11] M. Herlihy, "Atomic cross-chain swaps," in *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing*, pp. 245–254, Toronto, ON, Canada, July 2018.

[12] B. T. C. Relay, *Bridge between the Bitcoin blockchain and Ethereum smart contacts*, 2018.

[13] F. Vogelsteller and V. Buterin, "Eip 20: erc-20 token standard," *Ethereum Improvement Proposals*, vol. 20, 2015.

[14] S. Noether and B. Goodell, "Triptych:logarithmic-sized linkable ring signatures with applications," *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pp. 337–354, Springer, Berlin, Germany, 2020.

[15] J. Xie, *Nervos CKB: A Common Knowledge Base for Crypto-Economy*, 2018.

[16] J. Poon and T. Dryja, *The Bitcoin Lightning Network: Scalable Off-Chain Instant payments*, 2016.

[17] J. Kwon and E. Buchman, *Cosmos whitepaper*, 2019.

[18] G. Wood, *Polkadot: Vision for a Heterogeneous Multi-Chain framework*, White paper, vol. 21, no. 2327, 2016.

[19] Z. Liu, Y. Xiang, and J. Shi, "Hyperservice: interoperability and programmability across heterogeneous blockchains," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 549–566, London, UK, November 2019.

[20] H. Wang, Y. Cen, and X. Li, "Blockchain router: a cross-chain communication protocol," in *Proceedings of the 6th International Conference on Informatics, Environment*, pp. 94–97, energy and applications, New York NY, USA, August 2017.

[21] N. Szabo, "Smart contracts: building blocks for digital markets," *Extropy: The Journal of Transhumanist Thought*, vol. 18, no. 2, 1996.

[22] G. Wood, "Ethereum: a secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, pp. 1–32, 2014.

[23] E. Elrom, *Eos. Io Wallets and Smart Contracts*, The Blockchain Developer. Apress, Berkeley, CA, 2019.

[24] E. Androulaki, A. Barger, and V. Bortnikov, "Hyperledger fabric: a distributed operating system for permissioned blockchains," in *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–15, New York, NY, USA, July 2018.

[25] S. Thomas and E. Schwartz, *A Protocol for Interledger payments*, 2015.

[26] L. Gudgeon, P. Moreno-Sanchez, and S. Roos, "Sok: layer-two blockchain protocols," in *Proceedings of the International Conference on Financial Cryptography and Data Security (FC)*, pp. 201–226, New York, NY, USA, May 2020.

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

[28] A. Back, M. Corallo, and L. Dashjr, "Enabling blockchain innovations with pegged sidechains," vol. 72, pp. 201–224, 2014, http://www.opensciencereview.com/papers/123/enablingblockchaininnovations-with-pegged-sidechains.

[29] H. Abbas, M. Caprolu, and R. Di Pietro, "Analysis of polkadot: architecture, internals, and contradictions," in *Proceedings of the 2022 IEEE International Conference on Blockchain (Blockchain)*, pp. 61–70, Espoo, Finland, August 2022.