ATOM: Architectural Support and Optimization Mechanism for Smart Contract Fast Update and Execution in Blockchain-Based IoT

Tao Li, Yaozheng Fang, Zhaolong Jian, Xueshuo Xie, Ye Lu, and Guiling Wang

***Abstract*—Blockchain-based Internet of Things (BC-IoT) brings the advantages of blockchain into traditional IoT systems. In BC-IoT, the smart contract has been widely used for automatic, trusted, and decentralized applications. Smart contracts require frequent adjust and fast update due to various reasons, such as inevitable code bugs, changes of applications, or security requirements. However, previous smart contract architecture and updating mechanism are low speed and cause high overhead, because they are based on recompilation and redeployment in BC-IoT. Meanwhile, smart contract execution is so time consuming due to contract instruction dispatching and operand loading in the stack-based Ethereum virtual machine (EVM). To address these issues, we propose a new smart contract architecture and optimization mechanism for BC-IoTs, ATOM, which provides architectural supports to update contract economically and fast executing in instructionwise for the first time, to the best of our knowledge. We design a compact Application-oriented Instruction (AoI) set to describe application operations. We can construct the bytecode of smart contract from application by directly assembling templates prebuilt upon the AoIs rather than by compilation. We also present an optimized mechanism for AoI execution to enable access addressable storage place rather than the indirect access through stack. We perform ATOM on a BC-IoT testbed based on private Ethereum and Hyperledger Burrow. The experimental results highlight that ATOM is more efficient than state-of-the-art approaches. ATOM can reduce update latency by 62.7%, ledger size by 70%, and gas usage by 90% on average, respectively. Compared with the traditional smart contract architecture, ATOM can improve EVM Memory access efficiency significantly by up to 10× and achieve improvement of execution efficiency with up to 1.6×.**

Manuscript received March 8, 2021; revised May 20, 2021 and July 18, 2021; accepted August 19, 2021. Date of publication August 23, 2021; date of current version May 23, 2022. This work was supported by the National Key Research and Development Program of China under Grant 2018YFB2100300; in part by Zhejiang Lab under Grant 2021KF0AB04; in part by the Natural Science Foundation of Tianjin under Grant 20JCZDJC00610 and Grant 19JCQNJC00600; in part by the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant CARCHB202016 and Grant CARCH201905; and in part by the National Natural Science Foundation under Grant 62002175. *(Corresponding author: Ye Lu.)*

Tao Li and Ye Lu are with the College of Computer Science, Nankai University, Tianjin 300071, China, also with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China, and also with the Tianjin Key Laboratory of Network and Data Science Technology, Tianjin 300350, China (e-mail: luye@nankai.edu.cn).

Yaozheng Fang, Zhaolong Jian, and Xueshuo Xie are with the College of Computer Science, Nankai University, Tianjin 300071, China, and also with the Tianjin Key Laboratory of Network and Data Science Technology, Tianjin 300350, China.

Guiling Wang is with the Department of Computer Science, New Jersey Institute of Technology, Newark, NJ 07102 USA.

Digital Object Identifier 10.1109/JIOT.2021.3106942

***Index Terms*—Ethereum virtual machine (EVM), smart contract.**

I. Introduction

BLOCKCHAIN-BASED IoT (BC-IoT) is a new paradigm that uses blockchain to build distributed Internet of Things [1]–[6]. The paradigm has the advantages both of blockchain and IoT, e.g., trusted, decentralized, and tamper-proofing [7]–[9]. The BC-IoT has attracted extensive attention from both academia and industry [10], [11]. One of the most important parts of BC-IoT is smart contract, a computer program that can be automatically executed on blockchain such as Ethereum [12], and frees people from manual monitoring [13]–[15]. Owing to the attributes of autoexecution and consistent running result, the smart contract has been explored to enable many applications in BC-IoT [16]–[19], such as security management [20], [21].

Smart contracts in blockchain-based IoT system need to be updated frequently, because applications in such systems should be adjusted continuously for various reasons, such as inevitable code bugs and changes of application requirements [22]. For example, Luu *et al.* [23] and Huang *et al.* [24] pointed out that 8833 out of 19 366 existing Ethereum contracts are not bug free and vulnerable to attack. Thousands of smart contracts are potentially vulnerable, which should be corrected or patched up immediately through update [25]. However, the existing smart contract architecture in the original blockchain is not designed to support efficient contract update and execution. The most representative contract architecture and the most common-used contract execution environment are the Ethereum virtual machine (EVM). This work focuses on how to improve the contract update and execution performance on EVM.

The traditional contract update is based on recompilation and redeployment [26] in the application and the blockchain layer under the typical blockchain-based IoT [27], [28] as shown in Fig. 1. Updating a contract usually needs to take several steps as follows. First, smart contracts programmed by a high-level language (e.g., Solidity) are recompiled into the bytecode, which is composed of a series of contract instructions. In general, this recompilation requires long time and high memory footprint. Second, the bytecode needs to

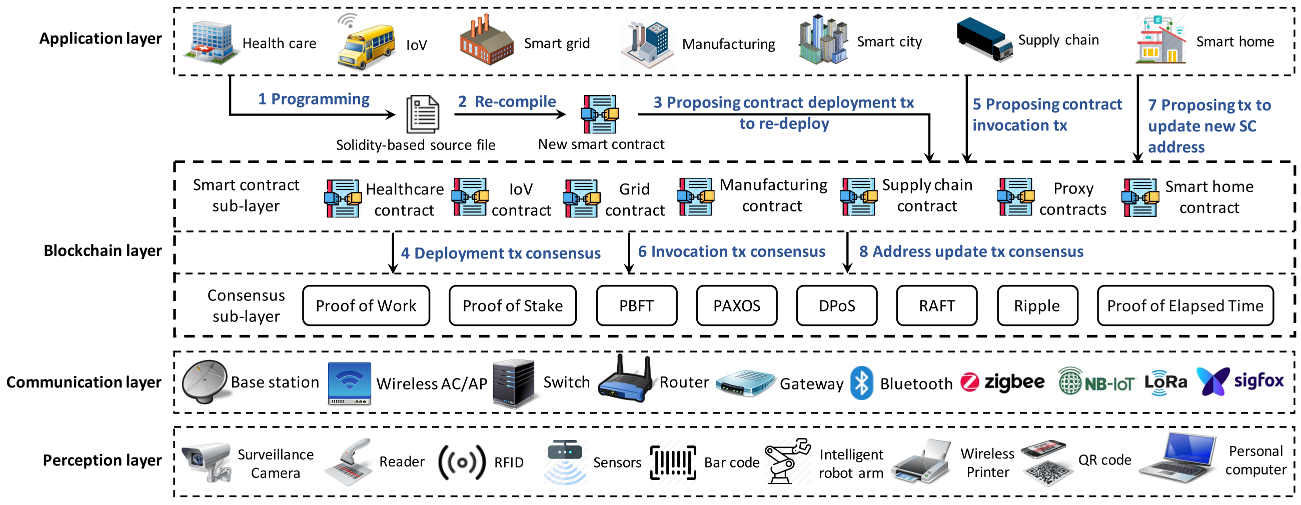


Fig. 1. Typical BC-IoT architecture.

be redeployed on the blockchain by proposing the *deployment transaction.* Until all nodes in the blockchain reach a consensus about this transaction, which takes a long time, the bytecode cannot be redeployed successfully. Finally, the redeployed bytecode of the updated smart contract can be executed by proposing another transaction named *invocation transaction* [29]. We can find that it is challenging to update the deployed contracts directly, which goes through tedious and time-consuming steps. There is no instruction support for bytecode modification to enforce the immutability of contracts or modify the bytecode directly. Consequently, a new smart contract must be recompiled into bytecode and redeployed on the blockchain, even with a few code modifications.

In fact, there are two remedial methods called the proxy-based method [30] and controller-data method, which can help to implement smart contract update, but their poor performance cannot meet the application requirements in blockchain-based IoT. Both of the two methods need to maintain an additional contract, such as proxy contract or controller contract, to record the address of each deployed smart contract. When deploying a new contract on the blockchain, the corresponding address needs to be updated in proxy or controller contract by broadcasting several transactions. Such indirect pattern introduces high storage overhead and large redundancy on the blockchain. Confirming these transactions usually take 20–60 s in blockchain-based IoT especially on the Ethereum. Moreover, updating the aforementioned 8833 contracts can take more than six months by the two methods. Such poor performance in the existing architecture has been proven to restrict smart contract to be widely applied in blockchain-based IoT. Therefore, the new architecture design for fast contract update should be explored deeply.

In addition, to efficiently update smart contracts, improving contract execution efficiency is another goal of this article. Because the stack-based virtual machine does not specify operand address, lots of operand loading instructions are generated, thus incurring high overhead. EVM [12], the current common smart contract execution environment, generates even more instructions for operand loading than normal stack-based virtual machine, because EVM *Stack* width is *fixed* [31], [32]. Specifically, EVM *Stack* is specially tailored for cryptographic computing: before loading operand to *Stack*, the hash value of the operands needs to be calculated. The hashing operation needs many EVM *Memory* access, thus causing high latency. Therefore, existing contract execution is not adaptive to applications with low-latency requirements. To solve this problem, BPU, a modularized architecture using FPGA, is proposed to accelerate smart contract execution [33]. However, such FPGA-based contract acceleration needs preliminary knowledge and has a long period for hardware design.

In order to get insights on how to improve contract update and execution efficiency, we first conduct a comprehensive performance analysis on contract parsing from the application (in particular, security policy), contract compilation, and execution. We learn that the time-consuming recompilation, interpretation, and data loading are the causes of the low efficiency. To this end, we propose a new advanced architecture and optimization mechanism named ATOM to support smart contract update and execution in instructionwise. In ATOM, a compact Application-oriented Instruction (AoI) set is presented to support contract update: the commonly used operations of a specific application are encapsulated into a few AoI instructions to replace dozens of EVM native instructions. ATOM constructs executable bytecode from the application by directly assembling templates prebuilt upon the AoIs rather than by compilation that has high overhead. This template-based method only needs source code parsing, but not complex morpheme or syntax analysis, which is an indispensable part of compilation. In ATOM, a new data access mechanism is also designed to enable directly loading the reference type operand from addressable data segment (e.g., EVM *Memory, Storage*) for AoI, while traditional mechanisms require all the operands to be loaded to EVM *Stack* first before calculation. This new mechanism in ATOM can decouple AoI execution data from *Stack* data in runtime, thus modifying AoI impacts

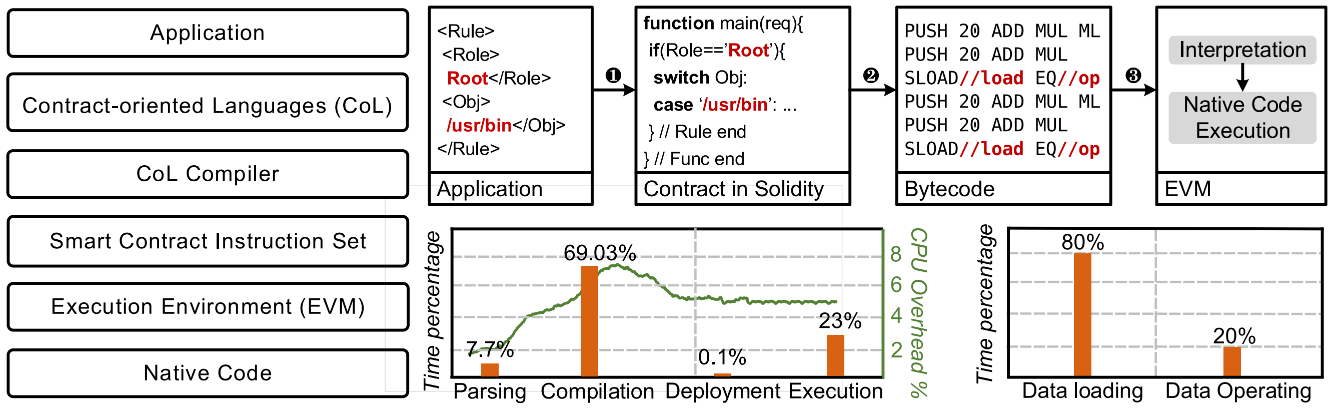

Application
Contract-oriented Languages (CoL)
CoL Compiler
Smart Contract Instruction Set
Execution Environment (EVM)
Native Code
<Rule> <Role> Root</Role> <Obj> /usr/bin</Obj> </Rule>
Application
❶
function main(req){ if(Role=='Root'){ switch Obj: case '/usr/bin': ... } // Rule end } // Func end
Contract in Solidity
❷
PUSH 20 ADD MUL ML PUSH 20 ADD MUL SLOAD//load EQ//op PUSH 20 ADD MUL ML PUSH 20 ADD MUL SLOAD//load EQ//op
Bytecode
❸
Interpretation
Native Code Execution
EVM
Time percentage
7.7%
69.03%
0.1%
23%
Parsing
Compilation
Deployment
Execution
CPU Overhead %
80%
20%
Data loading
Data Operating

Fig. 2. Hierarchical smart contract model and performance analysis. ❶ Parsing. ❷ Compilation. ❸ Execution.

on neither *Stack* nor EVM control flow. We summarize the key contributions of this article as follows.

1) We propose ATOM, a new smart contract architecture for fast updating and execution, and also validate ATOM efficiency through performance evaluation on our private blockchain-based IoT testbed.
2) We propose a template-based light-weighted bytecode construction mechanism that only involves application requirement parsing and template assembling rather than compilation. Our method can speedup bytecode construction by up to 10× compared with traditional compilation-based construction.
3) We present a compact AoI set (AoIS) that has strong descriptive capacity for application operations and decreases the number of bytecode instructions significantly. Compared with the state of the art, our instructions can reduce the update latency by up to 56%. The ledger size can be reduced by over 80% and the average GAS usage is reduced by 10%.
4) We develop contract executing optimization techniques to enable efficient AoI execution. The AoI operands are loaded in native code rather than pushed into the *Stack* before execution, which bypasses the heavy SHA3 computing and execution and decreases the EVM *Memory* access number by 90%. The experiment results highlight that ATOM can speedup the execution for more than 1.6×.

II. Background and Motivation

The typical architecture of smart contract has six layers, as shown in Fig. 2. In the top two layers, smart contracts are formulated based on application. In the two layers below, smart contracts described by contract-oriented language [34] (e.g., Solidity) are compiled into bytecode by a corresponding compiler in the third layer. Bytecode includes various types of contract instructions, such as stack operation, jump, and comparison. The bytecode is employed and executed in its own execution environment, such as EVM, and the native code layer.

A. Smart Contract Update

Most smart contracts were designed for different types of financial transactions [35], [36]. Because complex computing and frequent updating were not involved, the existing smart contract architecture was not designed to support efficient contract updating and execution. In particular, there is no instruction support to modify deployed bytecode directly. However, with the smart contracts rapid applying in many other fields, contract updating requires a higher level of frequency and timeliness.

Previous smart contract update can be realized by the proxy-based method, controller-data method, hot replace method, and fixed storage method to cope with contract immutability. The contract immutability, as aforementioned in Section I, refers that the records of contract modification cannot be tamper whatever but contract itself can be modified to a certain extent. The codes and the global variables in a contract are both stored as items in the key-value database, which can be updated or modified directly. Moreover, traditional EVM allows users or developers to modify the value of global variables.

The fixed storage method applies some fixed storage places to store the contracts' addresses before contract code running. When updating contract, the new contract's address is recorded in the certain fixed storage place. In fact, the two mainstream methods for updating smart contract are the proxy based and the controller data. There are also two kinds of contract in both of the two methods: 1) the business contract for application of the business execution and 2) proxy (or controller) contract for storing contracts' addresses. A business contract can be delegated by proxy or controller contract. When updating contract, the new programmed business contract must be compiled down to bytecode, and then the new bytecode should be deployed in blockchain. Overall, the processing for contract update in mentioned two methods is both realized through recompilation and redeployment [37].

Conducting contract updates require compilation-based bytecode construction but this procedure is very resource consuming, as shown in Fig. 2. Contract compilation time is approximately nine times longer than the parsing time, and compilation incurs high CPU overhead, which is twice higher than parsing. When frequent updating is needed in certain application scenarios, contract compilation becomes the bottleneck of the overall performance. Consequently, all of the above observations motivate us to modify contract from the bytecode level directly, in order to support contract fast update.

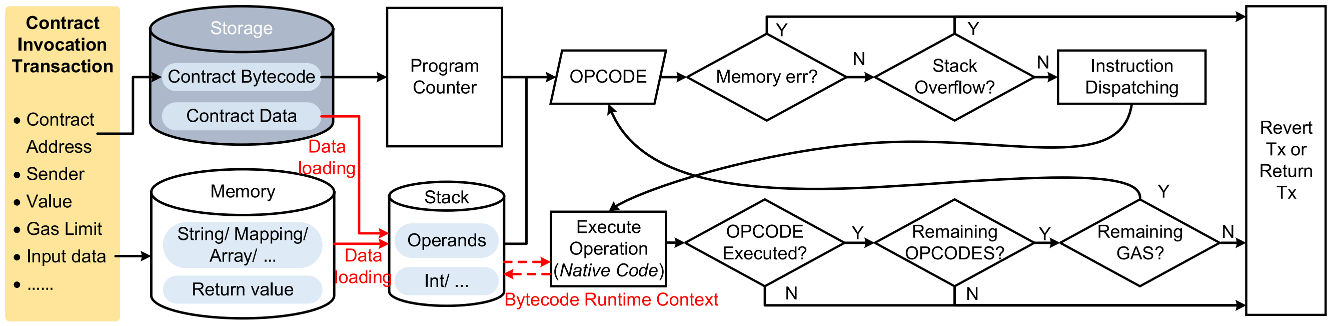


Fig. 3. Smart contract execution in EVM.

B. Smart Contract Execution

Contract execution can be expressed as follows [12]:

$$\boldsymbol{o} \equiv \Theta(T_d). \tag{1}$$

A smart contract function Θ can be invoked by a transaction [36]. The invocation transaction data T_d is the input of Θ. $\boldsymbol{o}$ is the output of Θ and it is a byte array.

For many applications, contract execution can be expressed in an extended form as follows:

$$\boldsymbol{o} \equiv \Theta(T_d, p) \tag{2}$$

$$\forall \boldsymbol{o},\ \exists A,\ \boldsymbol{o} \rightarrow A. \tag{3}$$

Here, p is parameter. Note that if p is empty, then function (2) becomes the same as (1). A is an action based on output $\boldsymbol{o}$. We use the role-based access control (RBAC) [38] as an example to illustrate the application. The RBAC rule checks whether the requester is in the accessible role list of the corresponding resource. T_d represents a specific access request, Θ represents the checking function, p represents the accessible role list, $\boldsymbol{o}$ is *FALSE* or *TRUE*, and the corresponding A is *DENY* and *ALLOW*. Such a smart contract can automatically allow or deny access requests. Smart contract updating is needed when the allowed role list changes or the access control rule is changed.

As shown in Fig. 3, Θ executes by stack-based EVM. Inside EVM, instruction operands and temporary fundamental variables are stored in *Stack*. Different from traditional virtual machines (e.g., Java Virtual Machine), the width of the *Stack* in EVM is fixed 256 bit, and the max depth of *Stack* is 1024 [31], [32]. T_d reference variables and function return value are stored in *Memory*. *Storage* is the online persistent storage. EVM follows Harvard architecture [12]: the contract bytecode and data are stored separately in *Storage*.

The execution of Θ follows four steps: 1) instruction retrieving; 2) storage place validation; 3) instruction dispatching; and 4) instruction execution. The first three steps are interpretation. According to the contract address in the invocation transaction, the program counter retrieves each instruction from *Storage*. Before instruction execution, EVM validates the *Memory* and *Stack* overflow. Finally, by instruction dispatching, instruction is executed in the native code layer. In such a stack-based environment, instruction operands need to be loaded into *Stack* before calculation. As a result, instruction native code execution can write/read the runtime context to/from *Stack*. Our preliminary studies show a surprising data that the validation and dispatching time take 99% of the whole execution time, while the execution in the native code layer only accounts for 1%. Furthermore, data loading time is up to four times the calculation time. This discovery tells us the validation time, dispatching time, and data-loading time should be reduced to improve system performance.

C. Limitations of Existing Architecture

Existing smart contract architecture cannot support economical contract update and fast execution for several reasons. First, existing contract instruction set cannot support contract update directly. There is no bytecode modification instruction that can be invoked to modify the code segment directly. Besides, instructions cannot be directly added or deleted in the compiled bytecode without incurring execution errors, such as jumping address invalidation. Instruction operands are stored in *Stack*. Adding or deleting instructions can impact the *Stack* context. This may break EVM control flow and cause jump address errors. EVM control flow is variables, not constants, which implies that the destination of a jump is a value read from *Stack* [31]. As a result, the contract only can be updated by redeployment.

Second, the existing execution optimization techniques cannot be applied to EVM. For example, two widely used traditional virtual machine execution optimization methods (Ahead-Of-Time [39] and just-in-time (JIT) compilation [40]) are not suitable for EVM. EVM can indeed achieve faster execution by ahead-of-time compilation (AOT) in certain scenarios. For example, some contracts that involve cryptographic computing have been precompiled for fast execution. But the AOT contracts are not flexible enough to support update, because precompiled contracts in the format of binary codes do not support runtime modification. The flexible JIT compilation may potentially support code update and faster execution. The JIT is used for dynamic programming languages, while the contract-oriented language, such as Solidity, is static. JIT is not suitable for smart contracts, which require data consistency among different nodes. JIT compiler may generate different compiled codes in different operating systems and hardware, which subsequently output different data and break data consistency.

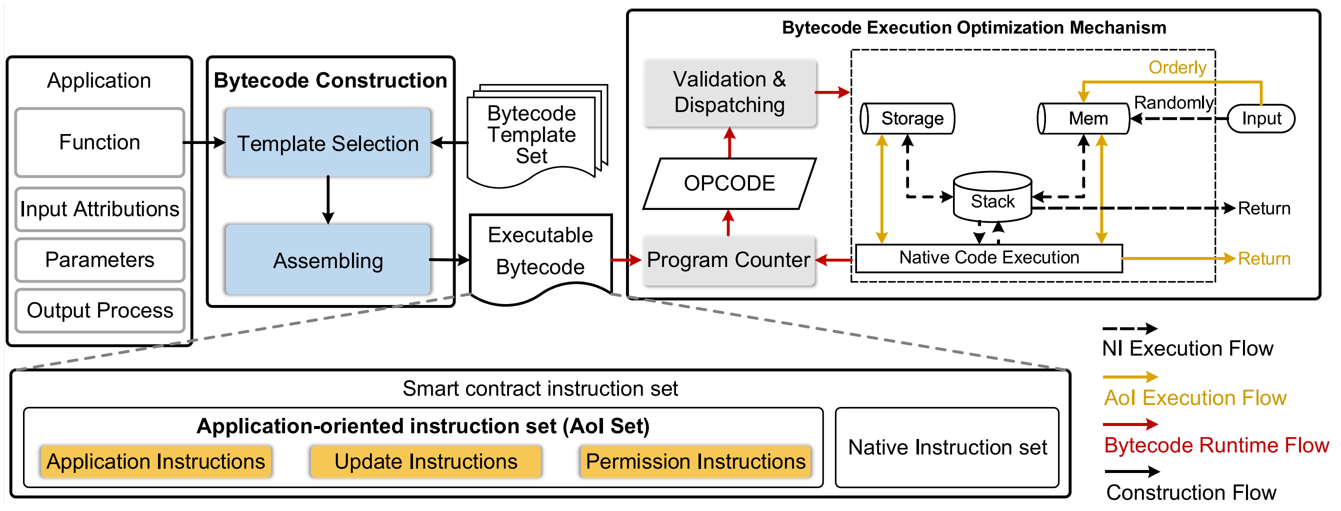


Fig. 4. ATOM overview.

III. ATOM Architecture

This article aims to achieve economical smart contract updating and efficient contract execution. Essentially, smart contract update is bytecode update. This motivates us to conduct smart contract update at instruction level for efficiency and provide holistic architecture support for contract update and execution. Our architecture design follows three guidelines.

1) The deployed bytecode should be modified directly to conduct the contract updating.
2) The number of bytecode instructions should be reduced as much as possible to shorten instruction dispatching and validation time, and thus, improve execution performance in EVM.
3) Contract recompilation should be removed as much as possible to reduce contract update time.

Following the above guidelines, we propose ATOM, application-oriented architecture support for efficient smart contract updating and execution for any application, which can be formulated by (2).

Fig. 4 shows an overview of ATOM. Given an application, ATOM employs a template-based bytecode construction mechanism to generate executable bytecode based on the EVM built-in native instruction set and an AoIS we designed. The AoIs will then be executed with execution optimization mechanisms we proposed. Specifically, the compact AoIS provides update and permission instructions and operation instructions commonly used in the application. AoIS greatly reduced the number of bytecode instructions by encapsulating many commonly used repetitive instructions into a few specific instructions. Update instructions are designed to support different types of updating, e.g., change of access control rules if the application is access control.

Based on the aforementioned AoIs and native instruction set, bytecode template sets are prebuilt based on the application. Then, the *template-based bytecode construction* constructs an executable bytecode by selecting a bytecode template from bytecode template set according to an application function, and assembles the selected template with other application data. This template-based construction can reduce compilation time and resource consumption.

An *execution optimization mechanism* is proposed to enable fast execution of AoIs by accessing *Memory* and *Storage* directly to reduce repetitive operand loading instructions. Note that AoI execution does not depend on *Stack* and thus, the AoI execution data are decoupled with *Stack* data in runtime. Hence, we are able to modify AoI directly to achieve contract update.

A. Application-Oriented Instruction Set

Extending the EVM native instruction set, we design a compact AoIS to support contract update and speedup contract execution. We add three types of instructions: 1) update instruction W_{updt} to enable contract update; 2) permission instruction W_{per} to manage whether to allow a contract updating or not; and 3) application instruction set W_{ao} to encapsulate the commonly used operations in the specific application. Before the AoI instruction execution, the corresponding operands will be pushed into Stack. When instruction executes, the operands are popped from Stack as parameters of the instruction execution.

Application instruction is used for executing application functions and operations. We take access control as an example to explain the design details about AoIS, as shown in Table I. Different access control models can utilize different instructions to realize their own function and action. For instance, the RBAC model needs to conduct role-based access checking frequently; once an access request is denied, the request to access certain resource should be blocked. Both functions are frequently used and involve many native instructions, which take a long time to run. Thus, we encapsulate them into ω_{RBAC} (`0x0d`) and ω_{DENY} (`0x1f`), respectively, to improve efficiency.

Update instruction is used for performing contract update. Contract updating can be either the data update of parameter p or the code update of Θ and A. We propose instructions ω_{DU}, ω_{PU}, and ω_{AU} to update data p, function Θ, and action A, respectively. For example, ω_{PU}(`0x0e`) represents code update,

TABLE I
EXAMPLES OF AoI IN ACCESS CONTROL

Type	Opcode	Name	Operand	Description
Application	`0x0d`	RBAC	Role, Source	Perform role-based checking with operands
	`0x0e`	ABAC	Attr, Source	Perform attribution-based checking with operands
	`0x0f`	SAC	Threshold	Perform simple request checking with operands
	`0x1f`	DENY	Source	Deny the requester
	`0x21`	ALLOW	Token	Return a file token
Update	`0x22`	DU	Loc, Value	Modify the value under specified location to realize data update
	`0x23`	PU	Addr, Value	Modify the byte under specified address to realize function update
	`0x24`	AU	Addr, Value	Modify the byte under specified address to realize action update
Permission	`0x25`	UR	Account	Specify the updatable account
	`0x26`	UT	Type Number	Specify the update type (`22, 23, 24` for `DU, PU, AU` accordingly)
	`0x27`	NOUPT0	Contract Address	Unenabled contract code update
	`0x27`	NOUPT1	Contract Address	Unenabled contract data update
Others	`0x28`	DMOV0	Loc, Value	Move the data from instruction to storage directly
	`0x29`	DMOV1	Loc, Value	Move the data from storage to instruction directly

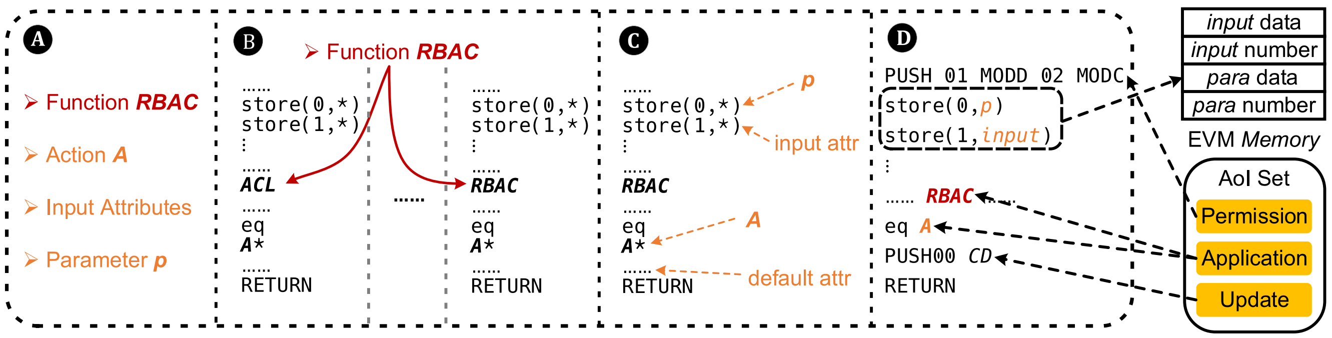


Fig. 5. Two-step bytecode construction. From left to right: Ⓐ Application, Ⓑ Template selection from template set, Ⓒ Assembling, and Ⓓ Executable bytecode.

which updates application function to ABAC, whose opcode is `0x0e`, as shown in Table I.

Upon receiving an update request ω_{PU} or ω_{DU}, ATOM first check whether the expected Θ or A exists in AoI or not. If they exist, the code is updated by ω_{PU} or ω_{AU}, respectively; otherwise, an error message is returned. Considering bytecode of Θ and A is stored in the code segment as a byte array, each byte represents an instruction identifier number or instruction operand. In ATOM, code update is achieved by modifying the byte value in the byte array.

Permission instruction is used to manage whether a contract update request can be granted or not. Permission instruction specifies which kind of roles can do contract update (e.g., owner or caller) and which kind of updates that different roles can execute (e.g., data update or code update). We design ω_{UR} and ω_{UT} to indicate the accounts that can do updating and the updatable types, respectively. During contract deployment, as predefined rules, the updatable roles and type information are written into a fixed place in *Storage* by ω_{UR} and ω_{UT}, respectively. Upon receiving an update request, ω_{UR} and ω_{UT} compare the update request information with the predefined update rules. Then, an agreement or rejection to the update request is returned.

B. Template-Based Bytecode Construction

In ATOM, we construct executable bytecode from application specification directly with the help of a precompiled bytecode template set. Bytecode construction has two steps: 1) template selection and 2) assembling.

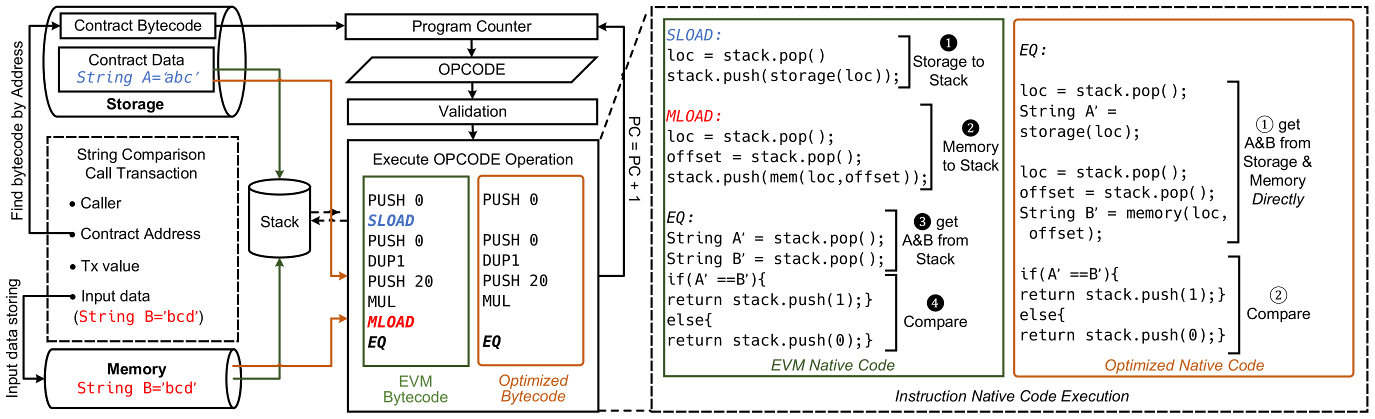
Building Bytecode Templates: For a specific application, we build bytecode templates based on different functions involved in this application. A bytecode template consists of two types of instructions: 1) AoI W_{ao} and 2) EVM native instruction W_{native}. W_{ao} can realize application functions while W_{native} can realize stack and control operations (e.g., push and jump), etc. A bytecode template T can be expressed as follows:

$$T := (W_{ao}, W_{\text{native}}) \subseteq \mathbb{T} \tag{4}$$

where $\mathbb{T}$ represents a bytecode template set.

Template Selection: We take access control application as an example (see Fig. 5). We list two bytecode templates in the template set: 1) template `ACL` and 2) template RBAC. Since the application specifies RBAC is needed, thus the corresponding template RBAC is selected, instead of template `ACL`.

Assembling: The template assembling needs two inputs: 1) the selected template and 2) application data (including



```
PUSH 0
SLOAD
PUSH 0
DUP1
PUSH 20
MUL
MLOAD
EQ
```

```
PUSH 0

PUSH 0
DUP1
PUSH 20
MUL

EQ
```

```
SLOAD:
loc = stack.pop()
stack.push(storage(loc));

MLOAD:
loc = stack.pop();
offset = stack.pop();
stack.push(mem(loc,offset));

EQ:
String A' = stack.pop();
String B' = stack.pop();
if(A' ==B'){
return stack.push(1);}
else{
return stack.push(0);}
```

```
EQ:

loc = stack.pop();
String A' =
storage(loc);

loc = stack.pop();
offset = stack.pop();
String B' = memory(loc,
 offset);

if(A' ==B'){
return stack.push(1);}
else{
return stack.push(0);}
```

Fig. 6. Execution optimization mechanism for string comparison.

input attributes, p and A). In addition, some default attributes (related to update and permission information) are also assembled in this step (if not specified in the application). In detail, T_d and p are assembled by W_{store} and A is assembled by W_A. After the assembling, the specific application is constructed into an executable bytecode.

Although Θ is certain in each executable bytecode, the number of T_d and p may be different, which is similar to function overload in programming. ATOM stipulates that T_d and p are stored in *Memory* rather than *Stack*, because the *Memory* is addressable. ATOM stores p first and T_d second. At the beginning of p segment and T_d segment, there is a flag to record the number of p and T_d.

C. Execution Optimization Mechanism

Bytecode execution includes operand loading and calculation. Reference type operands must be loaded from addressable storage place (e.g., *Memory* and *Storage*) into *Stack* before calculation. We use an example to illustrate the current contract execution procedure. As shown in Fig. 6, String A is stored in *Storage*. A call transaction is used for invoking a contract to compare string A with transaction input data (string B). Contract address in transaction refers that which contract is invoked. EVM can find the corresponding bytecode according to contract address. The input data of transaction are stored in *Memory* because string is a kind of reference type variable. For string comparison, EVM needs execute SLOAD and MLOAD instructions in EVM bytecode to load string from *Storage* and *Memory* to Stack accordingly (steps ❶ and ❷). When EQ instruction executes, the two strings need to be popped from *Stack* and be compared (steps ❸ and ❹). The instructions generated by these steps are fixed and repetitive.

In our optimized execution, optimized bytecode does not need SLOAD and MLOAD instruction to access *Memory* and *Storage* through *Stack* indirectly. Strings A and B can be accessed from *Storage* and *Memory* directly in the native code layer (①). Because instruction operands are stored in *Memory* orderly, we can leverage this knowledge to learn their address in advance. Then, the accessed strings can be compared directly (②). Our AoI execution does not rely on *Stack* to store operands, and thus, AoI modification has no impact on *Stack* data in runtime. As a result, the control flow will not be broken and we ensure the updating of the AoI has no errors.

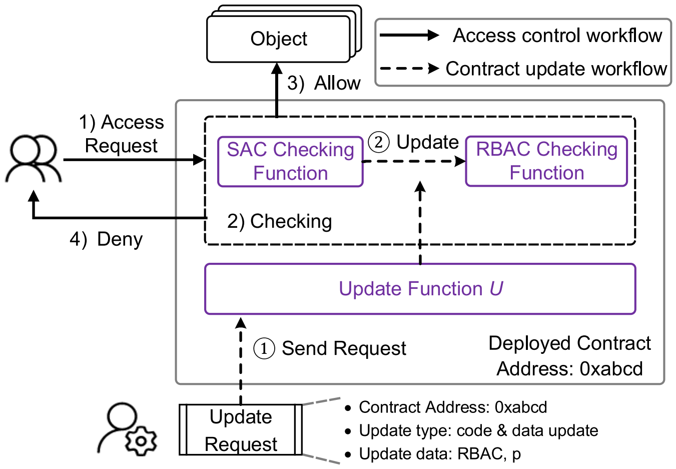


Fig. 7. Smart contract-based access control overview.

IV. Implementation: Access Control

Smart contracts have been deployed to implement automatic access control in BC-IoT [41], [42]. To evaluate the efficacy and efficiency of ATOM in supporting smart contract update and execution, we choose access control as the application to deploy smart contracts.

We implement two access control functions: 1) simple access control (SAC) [29], which can prevent Denial-of-Service (DoS) attack through comparing access interval with a preset threshold. Once a SAC contract is deployed, the threshold is fixed and 2) RBAC, which provides different access control to different roles, such as administrator and guest. Only roles in the accessible role list can conduct permitted operations, such as read or write, on certain resources. Once an RBAC contract is deployed, the accessible role list is fixed. The RBAC contract checks whether the role of the requester is inside the accessible role list. A contract update request can be either changing from SAC to RBAC and *vice versa* or changing the parameters of SAC and RBAC.

Fig. 7 shows an example of the smart contract-based access control. In the beginning, we deploy a smart contract that

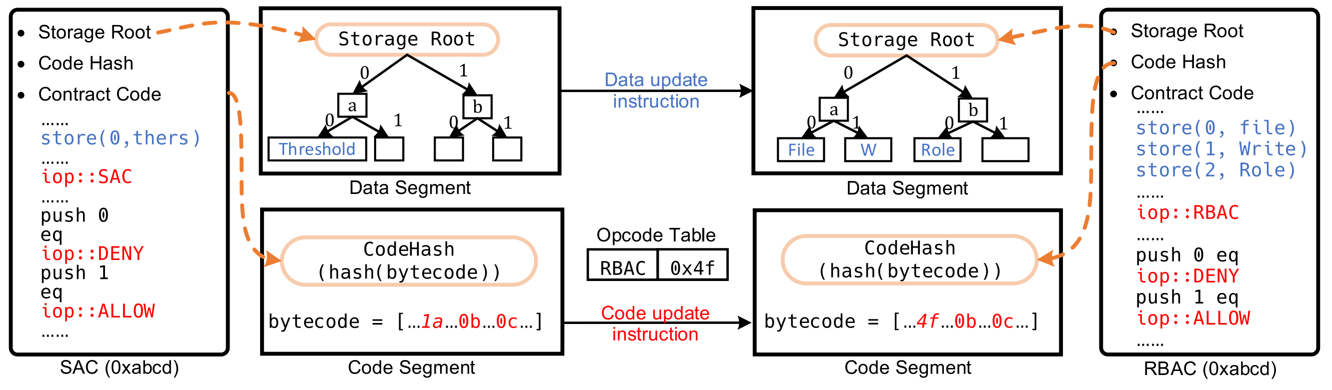


Fig. 8. Access control policy contract update process.

implements SAC. The corresponding bytecode is constructed by choosing SAC template and assembling it with other native node sets. This executable bytecode instance will then be deployed in the access control system. After deployment, the executable has a unique address, which is used for invoking. As shown in the left part of Fig. 8, the deployed executable bytecode has data segment and code segment. The data segment, which is the parameter of SAC contract, is organized as a trie, and SAC contract records a hash value of the trie as Storage Root. The bytecode of a SAC contract is stored as byte array, and the SAC contract records the hash value of the byte array. The SAC bytecode is deployed in the access control system with a unique address (0xabcd).

Later, we update the access control policy to be RBAC. Then, we need to update the smart contract accordingly. Note that we need to update both the code and the data considering RBAC has different parameters from SAC. As shown in Fig. 7, we first send an update request (step ①), which contains contract address, update type, and update data. Before the updating is executed, the update permission needs to be validated. ATOM first checks whether RBAC template exists in the template set or not and then conducts the data update and code update through update instructions (e.g., ω_{DU}) (step ②). Before and after the update, the address of contract is not changed. As shown in Fig. 8, for data update, the threshold is updated to the RBAC parameters (e.g., filename) in trie. For code update, ω_{SAC} is updated to ω_{RBAC}. Specifically, in the code segment, the instruction identity number of the corresponding position is modified from 0x1a to 0x4f. After contract update, the access request is checked by ω_{RBAC} rather than by ω_{SAC}. ATOM executes ω_{SAC} in the beginning and ω_{RBAC} after update both in the optimized execution mechanism.

V. Performance Evaluation

A. Evaluation Methodology

ATOM is evaluated on the application of access control in two real testbeds: 1) private Ethereum and 2) Hyperledger Burrow. The objectives of the evaluation are threefold: 1) testing the performance improvement of ATOM over traditional architecture regarding contract update and execution; 2) providing insights of ATOM's outperforming its peers; and 3) studying the impact of ATOM on the original Ethereum/ EVM.

To build the Ethereum (version 1.9.20) and Hyperledger Burrow (version 0.30.5) smart contract platforms, we employ eight nodes, three of which are equipped with ARMv7 CPU and 2GB memory, and five of which are equipped with XeonE5-2630 CPU (2.3 GHz, 6 Cores) and 96-GB memory. The nodes are connected via the same local area network. EVM (version 1.9.20) is the smart contract execution environment. The Ethereum official recommended evaluation framework Truffle[1] is employed to evaluate the performance. Smart contracts are coded in Solidity. We utilize XACML to implement the access control policy and the XML Reader[2] to parse the policy. In our experiments, we implemented two kinds of contracts: 1) SAC contract and 2) RBAC contract.[3] The smart contract is programmed by Solidity (version 0.5.1). We choose two widely adopted strategies for contract updating as our comparison baselines: 1) proxy-based and 2) controller-data model.[4] The source code for this evaluation is available at http://github.com/nkicsl/atom.

To evaluate the overall performance improvement, we choose the *latency* to accomplish the contract updating and *CPU overhead* throughout the whole updating procedure as the evaluation metrics. In addition, we choose *ledger size*, *gas usage*, and *EVM Memory overhead* to evaluate the overhead of contract updating incurred by ATOM and our peers.

B. Evaluation Results

1) Overall Improvement: In this section, we illustrate the performance of ATOM and our peers throughout the whole updating process: including parsing XACML to Solidity-described contract, contract compiling Solidity-described contract to executable bytecode, and bytecode update to eventually the bytecode execution. The parsing and contract compilation together are called bytecode construction. In ATOM, parsing refers to construct executable bytecode from XACML directly.

We randomly generate four groups of XACML policies. The policies are either SAC or RBAC. Each policy will be parsed into a smart contract. There are 500, 1000, 1500, and 2000

[1]https://www.trufflesuite.com/docs/truffle/

[2]https://www.npmjs.com/package/xmlreader/

[3]Note that currently there is no mature and widely accepted benchmark for rigorous evaluation of EVM performance. Although the real Ethereum transactions and contracts can be found on the public blockchain, it is unclear if such workload is sufficiently representative to EVM performance.

[4]https://github.com/fisco-bcos/fisco-bcos

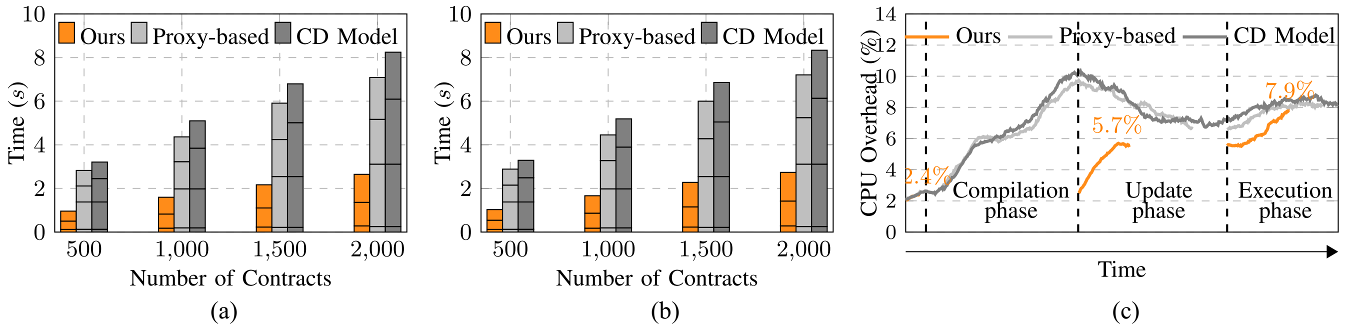


Fig. 9. Overall performance. (a) Latency in Ethereum. (b) Latency in Hyperledger Burrow. (c) CPU overhead in different phases.

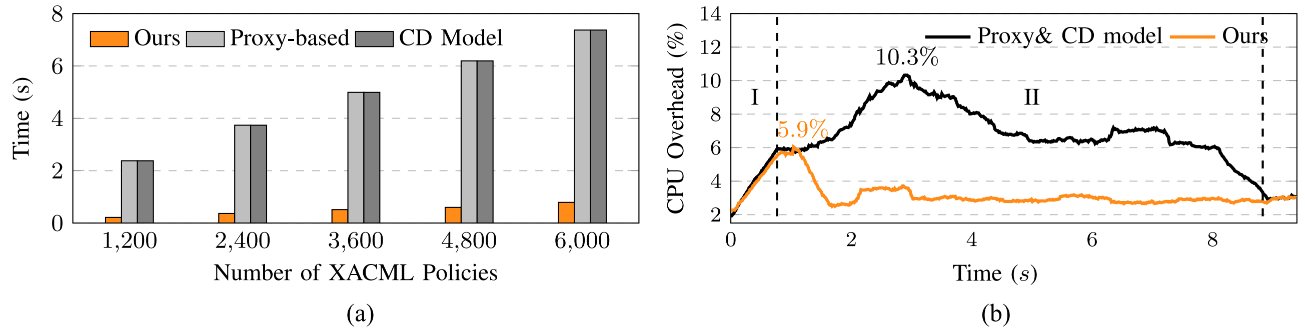


Fig. 10. Bytecode construction performance. (a) Bytecode construction latency. (b) CPU overhead (I: Parsing, II: Compilation).

XACML policies, respectively, for each group and correspondingly, the same number of smart contracts will be generated. Regarding the contract updating, the originally generated SAC smart contract will be updated to RBAC contracts and *vice versa*.

As shown in Fig. 9(a), in Ethereum, ATOM can reduce the overall latency by 63.9% and 68.7% compared with Proxy-based and CD model on average. As shown in Fig. 9(b), in Hyperledger Burrow, ATOM can reduce the latency by 62.7% and 67.6% on average. To illustrate the CPU overhead in each phase, we choose the group with 2000 policies (and contracts). As shown in Fig. 9(c), ATOM can reduce the CPU overhead in contract update and execution by 1.73% and 1.97% compared with Proxy-based and CD Model, respectively.

2) Bytecode Construction: For our comparison methods, bytecode construction refers XACML parsing and contract compilation. In our method, it refers XACML parsing, template selection, and assembling. We randomly generate five groups of XACML policies. There are 1200, 2400, 3600, 4800, and 6000 XACML policies, respectively, in each group. The function and parameter of XACML policy are generated randomly.

Fig. 10(a) illustrates the latency of bytecode construction time. We can see that ATOM can achieve 90% speedup than the baselines. For example, among the 6000 XACML policies, the average construction time of ATOM is 0.15 ms per policy, and our peers are about ten times more than ATOM. This is because, we construct bytecode by assembling selected templates rather than compilation. The template-based method only needs to parse the requirement attributions and assemble them with bytecode templates. On the contrary, the compilation needs to parse the contract source file, perform morpheme and syntax analysis, even if only one instruction is added or deleted.

Fig. 10(b) shows the CPU overhead during bytecode construction. ATOM has much lower CPU overhead. For example, ATOM CPU peak rate is 6% while the compilation peak rate is 10%. This is because the template-based bytecode construction does not need to perform the complex morpheme and syntax analysis as in compilation-based construction.

3) Contract Update: To evaluate contract updating, we prepare 12 groups of independent contracts. The number of contracts in each group ranges from 500 to 6000 with 500 as the increment. In each group, the function of each contract is random. Contract update is to modify the function of each contract to another function, i.e., we update all SAC contracts to RBAC contracts and *vice versa*.

Update latency is the time to accomplish contract update. (Note that the update latency does not include the blockchain system consensus time, considering the consensus time is influenced by many external factors, such as CPU capacity and mining difficulty, and has no direct relationship with ATOM.) Fig. 11(a) shows that in Ethereum, compared with the proxy-based update model, ATOM reduces the average update latency by about 47.8%. Compared with the controller-data model, ATOM reduces the average update latency by about 61.7%. Fig. 11(b) shows that the results in Hyperledger Burrow are 49.9% and 63.75%, respectively. This is because the controller-data model needs to perform index update not only in controller contract but also in data contract. The results prove that ATOM can update the access control contract in a shorter time, which adapts the fast dynamic system. Redeployment time is approximately equal to index update time, because they both process a transaction. Our peer

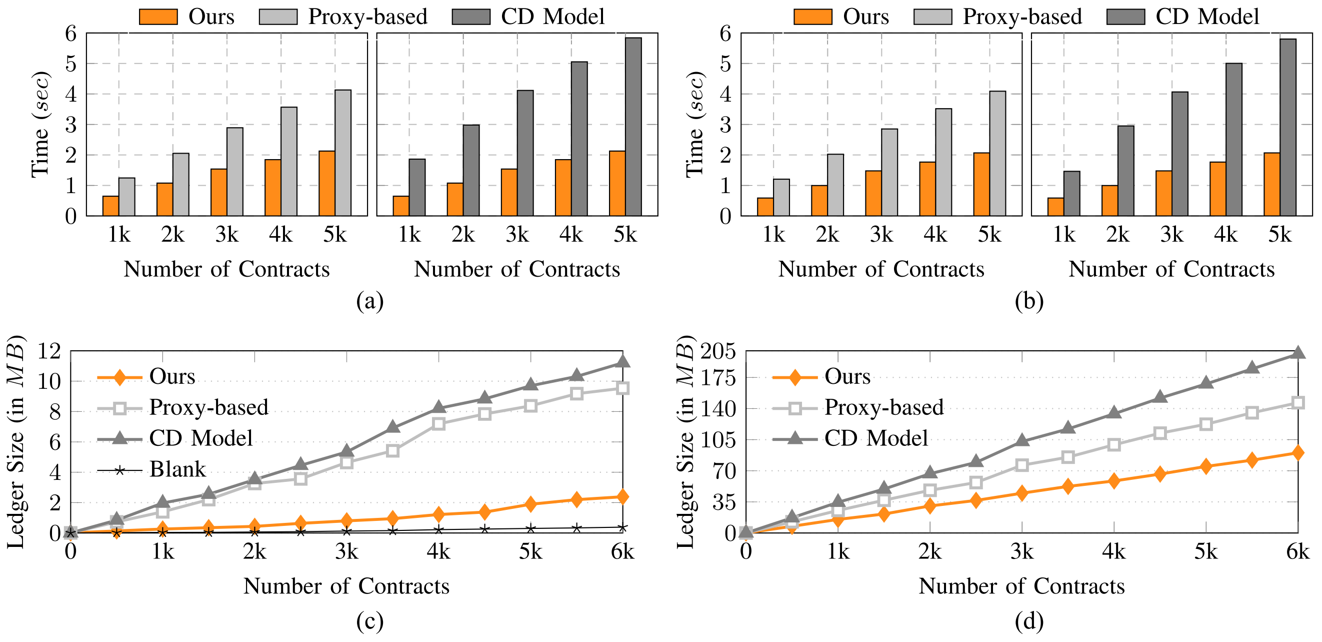


Fig. 11. Contract update consumption. (a) Contract update latency comparison in Ethereum. (b) Contract update latency comparison in Hyperledger. (c) Ledger size comparison in contract update in Ethereum. (d) Ledger size comparison in contract update in Hyperledger.

methods need to perform at least two transactions: 1) contract deployment and 2) index update, while ATOM only needs one transaction. In fact, the update at the instruction layer is similar to index update, because the update instruction also performs *Storage* data modification.

Ledger Size: In Ethereum, ledger size refers to the size of ledger file (/geth/chaindata/*.log) in runtime. In Hyperledger Burrow, ledger size refers to the log file (./*.log). Ledger data mainly include block data and user data (contract data, contract code, transaction message, etc.) [12]. Fig. 11(c) shows that in Ethereum, ATOM reduces the ledger size by about 81.3% and 84.1%, respectively, on average compared to our peer methods. In Hyperledger Burrow, Fig. 11(d) shows that the ledger size reduction achieves 39.4% and 55.6%, respectively. We also draw blank blockchain (pure block data) ledger size in Ethereum. After 6000 times update, ATOM is six times larger than the size of the blank blockchain, while the peer methods are 24 times and 29 times larger, respectively. The result implies that ATOM can be applied to systems with low storage capacity. ATOM achieves a smaller ledger because it can perform the update on the original contract directly rather than redeploying a new one. Note that those discarded smart contracts (data and code) cannot be deleted, so they are kept in the ledger permanently. The `Suicide` instruction provided by original EVM is used for disabling invocation but not for deleting contract data and code.

GAS Usage: Gas is the virtual currency used in Ethereum to measure the computational and storage resources required to perform certain actions on the Ethereum [32], [43]. For example, *ADD* instruction costs two units of gas, and *MUL* instruction costs three units. Less gas usage in contract updating and execution indicates a more economical and efficient method.

We compare gas usage in contract update regarding contract deployment and `SSTORE` execution, because our peer methods need both contract deployment and index update by `SSTORE` instruction. We price the gas usage in update instruction to be the same as `SSTORE` because both perform *Storage* modification. The gas usage is retrieved from the corresponding transaction receipt directly.

In this evaluation, we perform the contract update from SAC to RBAC ten times and calculate the average GAS usage. The results are shown in Table II: most gas are used for contract deployment in update in the two traditional methods since they need contract redeployment to achieve update. ATOM can save about 270 000 gas on average, because ATOM does not need contract deployment that we can modify code segment directly. The `SSTORE` gas usage is 20 000 (fixed) [12]. Traditional methods additionally need about 8000 gas usage for update parameter while ATOM needs about 6500 gas usage, considering the update parameter in ATOM is Int8 type (8 bit) rather than Ethereum Address type (160 bit).

4) Contract Execution (EVM Interpretation Latency): EVM interpretation includes instruction retrieving, storage place validation, and instruction dispatching. We prepare three contract functions: 1) an `IF` statement to perform string comparison. One string is stored in *Memory* and another is stored in *Storage*; 2) a `FOR` statement to perform repeated execution of `IF` statement; and 3) a recursive function to retrieve string from *Storage* for two times and then perform comparison. We execute the three functions equal number of times. ATOM provides three application instructions (ω_{if}, ω_{loop}, and ω_{recu}) to encapsulate `IF`, `FOR`, and recursive function, respectively.

The results show that ATOM can realize the aforementioned three functions by 101, 101, and 101 instructions, which includes one application instruction and 100 native

TABLE II
GAS USED COMPARISON IN CONTRACT UPDATE AMONG ATOM AND TWO TRADITIONAL METHODS

Method	Type	Gas Used ($\times 10^4$)	Reduced
Proxy-based	Deployment	26.2	Saved ✔
	Index Update	2.8	5.23 %
Controller-Data	Deployment	39.0	Saved ✔
	Index Update	2.8	5.46 %
ATOM	Instruction-layer Update	2.65	-

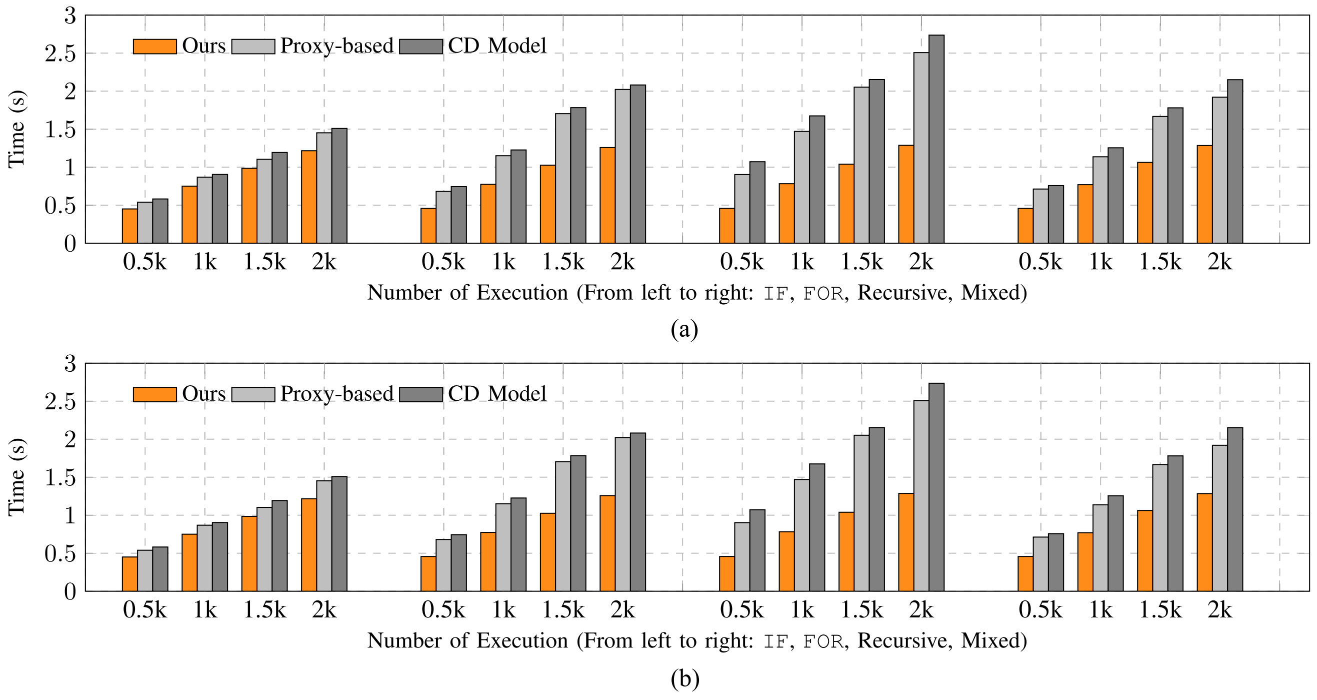


Fig. 12. Contract execution performance under different platforms and functions. (a) Contract execution performance under different functions in Ethereum. (b) Contract execution performance under different functions in Hyperledger Burrow.

TABLE III
CONTRACT EXECUTION LATENCY REDUCTION IN DIFFERENT PLATFORMS AND FUNCTION MODES

Baseline	Contract execution latency reduction (%)							
	Ethereum platform				Hyperledger Burrow platform			
	IF	FOR	Recursive function	Mixed execution	IF	For	Recursive function	Mixed execution
Proxy-based method	13.5	35.8	48.5	34.4	13.5	34.1	**47.5**	31.7
Controller-Data model	19.4	39.4	**53.8**	39.7	19.4	37.3	52.7	37.3

instructions. The proxy-based method needs 236, 814, and 1457 instructions and controller-data model needs 307, 923, and 1683 instructions to realize the three functions, respectively. We execute the three functions for 500, 1000, 1500, and 2000 times and calculate the average latency of ATOM and the peer methods. Fig. 12(a) shows that in Ethereum, ATOM can reduce 34.4% and 39.7% interpretation latency on average, compared with the proxy-based model and controller-data model, respectively. In Hyperledger Burrow, Fig. 12(b) shows that ATOM can reduce 37.3% and 31.7% interpretation latency on average (also can be seen in Table III).

Data Loading Latency: We deploy five smart contracts to evaluate the data loading latency with string loading. The smart contracts are initialized with a string array with different sizes (5000, 10 000, 15 000, 20 000, and 25 000). Each string is generated randomly, and the width of each string is less than the width of EVM Memory. We invoke the five contracts to load strings from *Memory* to *Stack*, and record loading time. Fig. 13(a) shows that ATOM can reduce the time for data loading by 76.7% on average. The time for different types of data loading is given in Fig. 13(b). We use flag `0` to represent loading data from *Storage* and flag `0` to represent loading data from *Memory*. Since ATOM loads the instruction operand from *Storage* and *Memory* directly, data loading latency can be reduced by 66.4% on average.

CPU Overhead: We choose string size 25 000 as an example to show the CPU overhead. As seen in Fig. 13(c), the time

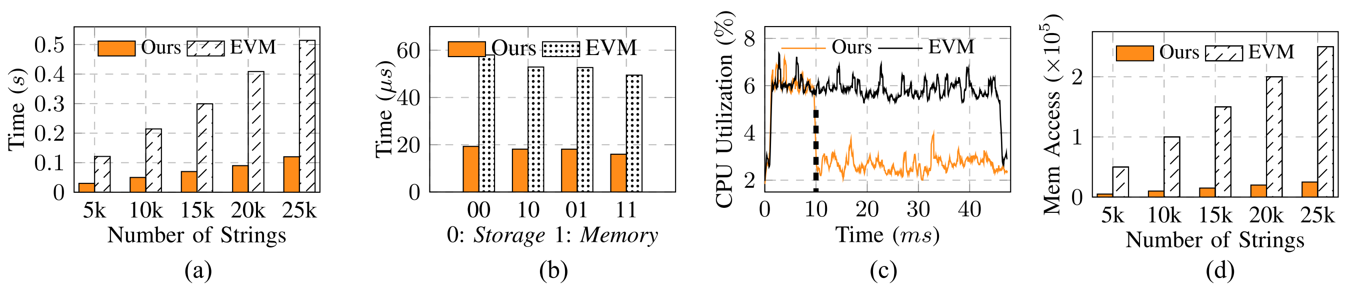


Fig. 13. Instruction execution performance comparison between ATOM and original EVM. (a) String loading time, from *Memory to Stack*. (b) String loading time in different storage situations. (c) CPU utilization under 25k times loading. (d) EVM Memory I/O.

for 25 000 times data loading in ATOM is 1–10 ms, while in EVM is 1–48 ms. During the data loading process, the average CPU overhead of ATOM is similar to EVM with a difference of 0.2%. Hence, ATOM does not bring any additional computation consumption.

EVM Memory Access: We evaluate the number of EVM *Memory* access during data loading by recording the number of `MLOAD` instruction execution. The results in Fig. 13(d) show that the number of *Memory* access is reduced by 90% in ATOM. ATOM does not need to execute `SHA3` instruction, which can make the width and type of operand be similar to *Stack* through hash calculation. `SHA3` needs a large number of *Memory* access to calculate the hash value. As a result, ATOM needs read *Memory* for only one time, while the original EVM needs ten times longer latency for one time data loading.

VI. Conclusion

In this article, we proposed ATOM, which provides architectural support for economical contract update and optimization mechanism for contract fast execution. We developed a compact AoIS to describe application operations, realize economical contract update, and propose template-based bytecode construction to realize lightweight construction from application to bytecode. We also designed an execution optimization mechanism for AoI to realize fast execution and decouple AoI execution data from *Stack* data in runtime. The experimental results highlight that ATOM has greatly outperform the state-of-the-art update methods, and our architecture expectedly matches the requirements of the blockchain-based IoT.

References

[1] C. Lin, D. He, N. Kumar, X. Huang, P. Vijayakumar, and K.-K. R. Choo, "HomeChain: A blockchain-based secure mutual authentication system for smart homes," *IEEE Internet Things J.*, vol. 7, no. 2, pp. 818–829, Feb. 2020.

[2] J. Pan, J. Wang, A. Hester, I. Alqerm, Y. Liu, and Y. Zhao, "EdgeChain: An edge-IoT framework and prototype based on blockchain and smart contracts," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4719–4732, Jun. 2019.

[3] O. Novo, "Scalable access management in IoT using blockchain: A performance evaluation," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4694–4701, Jun. 2019.

[4] B. K. Mohanta, D. Jena, S. Ramasubbareddy, M. Daneshmand, and A. H. Gandomi, "Addressing security and privacy issues of IoT using blockchain technology," *IEEE Internet Things J.*, vol. 8, no. 2, pp. 881–888, Jan. 2021.

[5] Y. Qu *et al.*, "Decentralized privacy using blockchain-enabled federated learning in fog computing," *IEEE Internet Things J.*, vol. 7, no. 6, pp. 5171–5183, Jun. 2020.

[6] F. Chen, Z. Xiao, L. Cui, Q. Lin, J. Li, and S. Yu, "Blockchain for Internet of Things applications: A review and open issues," *J. Netw. Comput. Appl.*, vol. 172, Dec. 2020, Art. no. 102839.

[7] H.-N. Dai, Z. Zheng, and Y. Zhang, "Blockchain for Internet of Things: A survey," *IEEE Internet Things J.*, vol. 6, no. 5, pp. 8076–8094, Oct. 2019.

[8] M. A. Ferrag, M. Derdour, M. Mukherjee, A. Derhab, L. Maglaras, and H. Janicke, "Blockchain technologies for the Internet of Things: Research issues and challenges," *IEEE Internet Things J.*, vol. 6, no. 2, pp. 2188–2204, Apr. 2019.

[9] H. Wang, D. He, J. Yu, N. N. Xiong, and B. Wu, "RDIC: A blockchain-based remote data integrity checking scheme for IoT in 5G networks," *J. Parallel Distrib. Comput.*, vol. 152, pp. 1–10, Jun. 2021.

[10] Y. Xu, J. Ren, G. Wang, C. Zhang, J. Yang, and Y. Zhang, "A blockchain-based nonrepudiation network computing service scheme for industrial IoT," *IEEE Trans. Ind. Informat.*, vol. 15, no. 6, pp. 3632–3641, Jun. 2019.

[11] M. Wu, K. Wang, X. Cai, S. Guo, M. Guo, and C. Rong, "A comprehensive survey of blockchain: From theory to IoT applications and beyond," *IEEE Internet Things J.*, vol. 6, no. 5, pp. 8114–8154, Oct. 2019.

[12] G. Wood. (2014). *Ethereum: A Secure Decentralised Generalised Transaction Ledger*. [Online]. Available: http://gavwood.com/Paper.pdf

[13] C. Ge, Z. Liu, and L. Fang, "A blockchain based decentralized data security mechanism for the Internet of Things," *J. Parallel Distrib. Comput.*, vol. 141, pp. 1–9, Jul. 2020. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S074373151930810X

[14] L. Zhu, C. Chen, Z. Su, W. Chen, T. Li, and Z. Yu, "BBS: Micro-architecture benchmarking blockchain systems through machine learning and fuzzy set," in *Proc. IEEE Int. Symp. High Perform. Comput. Architect. (HPCA)*, 2020, pp. 411–423.

[15] Y. Zhang and J. Wen, "An IoT electric business model based on the protocol of bitcoin," in *Proc. 18th Int. Conf. Intell. Next Gener. Netw.*, 2015, pp. 184–191.

[16] U. Majeed, L. U. Khan, I. Yaqoob, S. A. Kazmi, K. Salah, and C. S. Hong, "Blockchain for IoT-based smart cities: Recent advances, requirements, and future challenges," *J. Netw. Comput. Appl.*, vol. 181, May 2021, Art. no. 103007.

[17] J. Chen, S. Yao, Q. Yuan, K. He, S. Ji, and R. Du, "CertChain: Public and efficient certificate audit based on blockchain for TLS connections," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2018, pp. 2060–2068.

[18] S. Hu, C. Cai, Q. Wang, C. Wang, X. Luo, and K. Ren, "Searching an encrypted cloud meets blockchain: A decentralized, reliable and fair realization," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2018, pp. 792–800.

[19] R. Cheng *et al.*, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *Proc. IEEE Eur. Symp. Security Privacy (EuroS&P)*, 2019, pp. 185–200.

[20] Z. Su, Y. Wang, Q. Xu, M. Fei, Y.-C. Tian, and N. Zhang, "A secure charging scheme for electric vehicles with smart communities in energy blockchain," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4601–4613, Jun. 2019.

[21] Y. Zhang, M. Yutaka, M. Sasabe, and S. Kasahara, "Attribute-based access control for smart cities: A smart contract-driven framework," *IEEE Internet Things J.*, vol. 8, no. 8, pp. 6372–6384, Apr. 2021.

[22] J. Xu *et al.*, "Healthchain: A blockchain-based privacy preserving scheme for large-scale health data," *IEEE Internet Things J.*, vol. 6, no. 5, pp. 8770–8781, Oct. 2019.

[23] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2016, pp. 254–269.

[24] Y. Huang, Q. Kong, N. Jia, X. Chen, and Z. Zheng, "Recommending differentiated code to support smart contract update," in *Proc. IEEE/ACM 27th Int. Conf. Program Comprehension (ICPC)*, 2019, pp. 260–270.

[25] P. Bailis, A. Narayanan, A. Miller, and S. Han, "Research for practice: Cryptocurrencies, blockchains, and smart contracts; hardware for deep learning," *Commun. ACM*, vol. 60, no. 5, pp. 48–51, 2017.

[26] Z. Zheng *et al.*, "An overview on smart contracts: Challenges, advances and platforms," *Future Gener. Comput. Syst.*, vol. 105, pp. 475–491, Apr. 2020.

[27] S. Biswas, K. Sharif, F. Li, B. Nour, and Y. Wang, "A scalable blockchain framework for secure transactions in IoT," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4650–4659, Jun. 2019.

[28] M. Zhaofeng, M. Jialin, W. Jihui, and S. Zhiguang, "Blockchain-based decentralized authentication modeling scheme in edge and IoT environment," *IEEE Internet Things J.*, vol. 8, no. 4, pp. 2116–2123, Feb. 2021.

[29] Y. Zhang, S. Kasahara, Y. Shen, X. Jiang, and J. Wan, "Smart contract-based access control for the Internet of Things," *IEEE Internet Things J.*, vol. 6, no. 2, pp. 1594–1605, Apr. 2019.

[30] W. Shao, Z. Wang, X. Wang, K. Qiu, C. Jia, and C. Jiang, "LSC: Online auto-update smart contracts for fortifying blockchain-based log systems," *Inf. Sci.*, vol. 512, pp. 506–517, Feb. 2020.

[31] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: Thorough, declarative decompilation of smart contracts," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng. (ICSE)*, 2019, pp. 1176–1186.

[32] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: Surviving out-of-gas conditions in ethereum smart contracts," in *Proc. ACM Program. Lang.*, vol. 2, 2018, pp. 1–27.

[33] T. Lu and L. Peng, "BPU: A blockchain processing unit for accelerated smart contract execution," in *Proc. 57th ACM/IEEE Design Autom. Conf. (DAC)*, 2020, pp. 1–6.

[34] A. Juels, A. Kosba, and E. Shi, "The ring of gyges: Investigating the future of criminal smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2016, pp. 283–295.

[35] P. Cuccuru, "Beyond bitcoin: An early overview on smart contracts," *Int. J. Law Inf. Technol.*, vol. 25, no. 3, pp. 179–195, 2017.

[36] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town crier: An authenticated data feed for smart contracts," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2016, pp. 270–282.

[37] M. Lohr and S. Peldszus, "Maintenance of long-living smart contracts," in *Proc. Workshops Softw. Eng. (SE)* , Mar. 2020, pp. 98–104.

[38] A. Saini, Q. Zhu, N. Singh, Y. Xiang, L. Gao, and Y. Zhang, "A smart contract based access control framework for cloud smart healthcare system," *IEEE Internet Things J.*, vol. 8, no. 7, pp. 5914–5925, Apr. 2021.

[39] N. Reijers and C.-S. Shih, "Improved ahead-of-time compilation of stack-based JVM bytecode on resource-constrained devices," *ACM Trans. Sensor Netw.*, vol. 15, no. 3, pp. 1–44, Aug. 2019.

[40] E. Schkufza, M. Wei, and C. J. Rossbach, "Just-in-time compilation for verilog: A new technique for improving the FPGA programming experience," in *Proc. ASPLOS*, 2019, pp. 271–286.

[41] B. Bera, S. Saha, A. K. Das, and A. V. Vasilakos, "Designing blockchain-based access control protocol in IoT-enabled smart-grid system," *IEEE Internet Things J.*, vol. 8, no. 7, pp. 5744–5761, Apr. 2021.

[42] J. Qiu, Z. Tian, C. Du, Q. Zuo, S. Su, and B. Fang, "A survey on access control in the age of Internet of Things," *IEEE Internet Things J.*, vol. 7, no. 6, pp. 4682–4696, Jun. 2020.

[43] X. L. Yu, O. Al-Bataineh, D. Lo, and A. Roychoudhury, "Smart contract repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 4, pp. 1–32, 2020.