



## Research Article

# To Delay Instantiation of a Smart Contract to Save Calculation Resources in IoT

Hong Su,<sup>1</sup> Bing Guo ,<sup>1</sup> Yan Shen,<sup>2</sup> Zhen Zhang,<sup>1</sup> and Chaoxia Qin <sup>1</sup>

<sup>1</sup>College of Computer Science, Sichuan University, Chengdu 610041, China

<sup>2</sup>Control Engineering College, Chengdu University of Information Technology, Chengdu 610041, China

Correspondence should be addressed to Bing Guo; guobing@scu.edu.cn

Received 20 October 2020; Revised 10 January 2021; Accepted 4 February 2021; Published 18 February 2021

Academic Editor: Hongju Cheng

Copyright © 2021 Hong Su et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Smart contracts are required to be instantiated in the predeployed stage, which consumes computation resources from then on. It is a big waste in the blockchain whose nodes are composed of IoT devices, as those devices often have limited resources (such as limited power supplies or a limited number of processes to run). Meanwhile, IoT devices are heterogeneous and different smart contracts are required. If those smart contracts are instantiated previously, numerous meaningless addresses are required. In this paper, we propose to delay the instantiation of a smart contract when used and terminate it when not used, which is similar to the life cycle of a variable. Then, a new kind of variable (the wrapping variable) is used to hide details of the instantiation and the address. The smart contract is instantiated in the construction function of the wrapping variable, or even it is delayed to the time when there are requests for it. The smart contract terminates when the variable is out of its scope. Then, different instantiation methods are proposed. Finally, we perform the qualitative comparison between the proposed approach and the predeployment method, and it demonstrates that the proposed methods optimize the life cycle of the smart contract and save calculation resources.

## 1. Introduction

Smart contracts are widely used in the cooperation between blockchains and IoT devices, with the aim of achieving reliability, security, and trust [1, 2]. To manage numerous IoT devices, smart contracts are used to control and configure IoT devices in a secure way [3]. For the efficient data aggregation [4] in IoT, [5] proposes an aggregation way with privacy preserved. As intrusion detection is important in IoT [6, 7], a collaborative intrusion detection based on smart contracts is proposed [8]. For the interaction with the nonblockchain system (like IoT), [9] proposes a direct interaction way without an oracle [10].

Currently, smart contracts are required to be deployed previously [11], which preinstantiates the smart contracts [12, 13]. The instance of a smart contract occupies calculation resources (including the memory, CPU, or disk space) of blockchain nodes. Even when there is no request for a

smart contract, its instance is not terminated and the resources are occupied. This is suitable in the blockchain whose nodes have external power supplies (such as a PC or a server), while blockchain nodes may be composed of IoT devices [14]. The resources of IoT devices are often limited [15, 16], such as limited power supply or a limited number of processes to run. Then, it is fatal to delay or reduce requests for a smart contract to save power.

Meanwhile, a unique address [17, 18] or identifier is required to access a smart contract in the predeployment method. IoT devices are of big amount (more than 30 billion in 2020 (<https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>)) and various types, which results in various smart contracts. If we instantiate those smart contracts previously, users have to use numerous meaningless addresses to interact with blockchains. Some deployment tools [19] or platforms [20, 21] facilitate the deployment process. However, this process

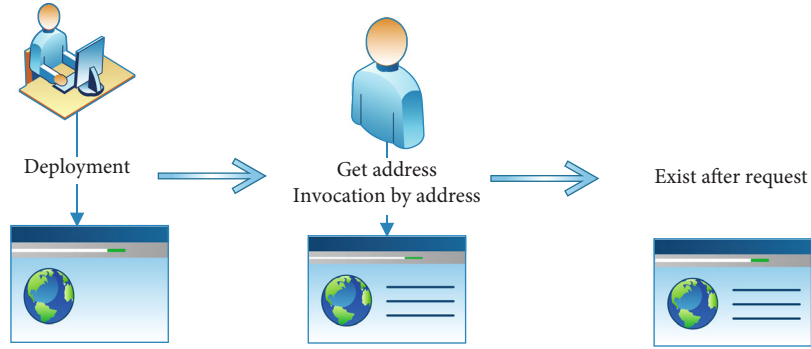


FIGURE 1: Current process of smart contracts. The smart contract is instantiated previously, and an address is returned. The caller interacts with the smart contract by the address. In this way, a smart contract exists even if there is no request.

makes the instantiation details being exposed to the callers (smart contract author or its user) and forces the callers to use the addresses of smart contracts. Figure 1 shows an example.

In this paper, we propose the methods to delay the instantiation of a smart contract after the smart contract has been deployed and to perform the destruction of a smart contract timely to save calculation resources. This way is similar to the life cycle management of a local variable. Then, we propose to use variables to wrap the instantiation steps of smart contracts. The details of the deployment of a smart contract are not exposed to the caller, which is performed in the background.

The major contributions of this paper are as follows.

- (1) We propose to create a smart contract instance when there are requests for it and terminate the instance when it is not used (out of its usage scope), which is similar to a local variable. It is aimed at delaying the instantiation of a smart contract and terminating a smart contract in time
- (2) We propose to adopt a new kind of variable to hide the details of the instantiation and termination. This kind of variety is called the wrapping variable. The caller only needs to create a wrapping variable, and in its construction, the smart contract is instantiated. When the variable is out of its scope, the smart contract terminates. It only needs to control a variable to optimize the life cycle of a smart contract. This method also facilitates one smart contract to invoke another smart contract, in which the latter is a local variable
- (3) We propose two instantiation methods: instant instantiation and postinstantiation. Instant instantiation creates a smart contract instance when its wrapping variable is created. Postinstantiation creates the smart contract instance when there are real invocations to the wrapping variable

The rest of this paper is organized as follows. Section 2 describes different instantiation methods. Section 3 shows the simulation results and the corresponding analysis. Section 4 gives the summary and concludes the paper.

## 2. Instantiation Methods

**2.1. Motivation.** Currently, the smart contract instantiation is done before the execution, and this procedure is called deployment. It has some disadvantages as described in Introduction. However, a smart contract can be used as the way that a variable is used—it is dynamically instantiated and terminated. It shortens the runtime of a smart contract. In this method, the instantiation is performed when there are invocations of a smart contract. When it is not used or out of its usage scope, the termination of a smart contract is performed. It also facilitates the usage of a smart contract to use a variable instead of the global address of the deployed smart contract.

**2.2. Different Instantiation Methods.** We can instantiate an instance beforehand or delay the instantiation when there is a function call (invocation) to the smart contract. Figure 2 shows three possible instantiation methods.

**2.2.1. Preinstantiation Method.** This is the currently used method, in which the instantiation is performed before the real execution. The instance is associated with an address (or other unique identifiers), which is used to look up the smart contract instance and dispatch corresponding invocations to it. However, this instance may run without any invocation for a long time (in a waiting state). When there are no invocations, the smart contract still resides in the memory until it has been obviously been terminated. It is shown in the “preinstantiation” part of Figure 2.

**2.2.2. Instant Instantiation.** In this method, a smart contract is instantiated when it has been called by another smart contract or a user. (1) If smart contract  $T$  is invoked by smart contract  $S$ ,  $S$  creates a variable  $V$  which wraps  $T$ . Variable  $V$  instantiates smart contract  $T$  in its constructor. Successive access to the instance of  $T$  is through variable  $V$ . (2) If it is called by a user, the instantiation is triggered by the blockchain platform when its user sends a transaction to trigger its function. Then, the request is passed to the newly created instance. In both cases, there is no need to give an address to the caller. Then, a new kind of variety is proposed to access the smart contract instance. In this paper, we define a new class (Java class) to stand for this kind of variable.

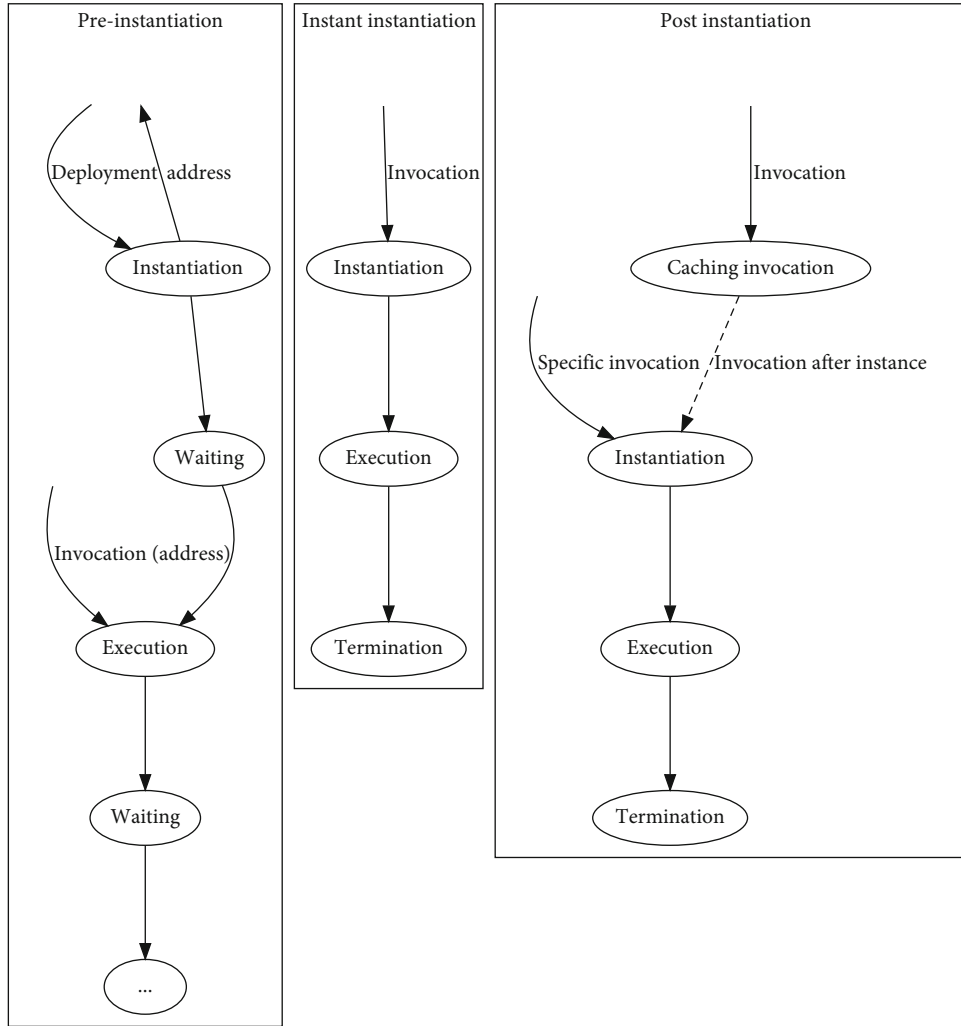


FIGURE 2: Different instantiation methods.

The instance is terminated when the variable is not required; either the calling smart contract exits or the variable is out of its scope. The scope of a variable further shortens the runtime of a smart contract instance. If we put the variable into a smaller scope, the variable lives only in this scope. There is a difference between the normal variable termination and the variable for the smart contract. The latter one requires that the execution results of the smart contract have been confirmed in the blockchain. After the confirmation, the blockchain consensus protocol enforces the correctness of this smart contract [14].

For other methods of the smart contract, the corresponding variable forwards it to the smart contract. As the smart contract is running in a different process, the method invocation is done by the interprocess communication (IPC) on each node, while this is hidden in the implementation of the variable. Figure 3 shows the relationship between the variable and the smart contract.

Instant instantiation facilitates the invocation of different smart contracts. The caller does not need to care about the instantiation and termination of a smart contract. Further, the caller is not required to access a smart contract by a string

of meaningless numbers (the address of a smart contract in the predeployment method). Algorithm 1 is an example.

From Algorithm 1, we know that instances of smart contracts SCA and SCB are created and sealed in variables at lines 12 and 16, and then, their methods are accessed by those two variables. Those variables are automatically deconstructed at the end of this scope (line 18).

**2.2.3. Postinstantiation (Lazy Instantiation).** When some methods of a smart contract are invoked, the instance of the smart contract does not have to be created. For example, a caller updates a value to a smart contract continuously and no users retrieve this value. During this time, it is not necessary to have an instance. We can cache those updating methods in the variable and only instantiate the smart contract when a user tries to get the value. Figure 4 shows this process.

Postinstantiation is that a smart contract is instantiated when there is a request that its instance has to be created. The corresponding request is called the instantiation must request or method (IMR), and other requests are called the instantiation unnecessary request or method (IUR).

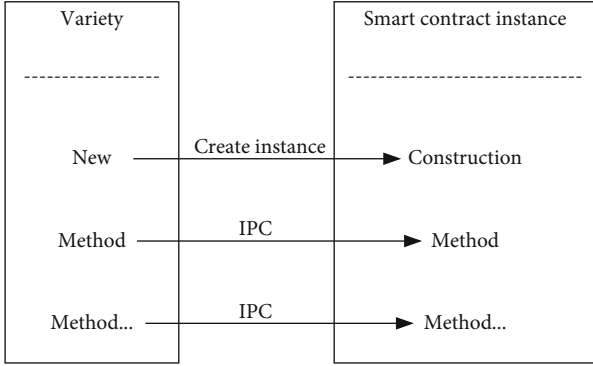


FIGURE 3: The relationship between the variable and the smart contract instance. The interaction among the variable and the smart contract instance is the IPC (interprocess communication) method.

Postinstantiation is aimed at delaying the instantiation of the smart contract. IUR requests sent before IMR are cached. Those logics are inside the variable, which is illustrated in Algorithm 2.

The postinstantiation delays the instantiation before IMR. And if the user has not sent any IMR, the instance can even be avoided. In this case, it is still traceable, as the transaction to invoke the smart contract is stored in the blockchain.

**2.3. Resource Analysis.** In this section, we analyze the resources (CPU and memory) saved by the dynamical instantiation and termination. We regard a smart contract ( $S$ ) as a collection of smart contract copies ( $s$ ) running on different nodes. The notation of  $s_i$  denotes the smart contract copy ( $s$ ) on different node  $i$  shown as follows.

$$S = [s_1, \dots, s_k, \dots]. \quad (1)$$

When the smart contract is instantiated, it occupies corresponding resources, such as memory and CPU. We use  $rs_i$  to denote its resources on node  $i$ .

$$rs_i = [\text{memory}, \text{CPU}, \dots]. \quad (2)$$

Node resources are occupied during the time from the instantiation to the termination. We use  $rs_i \sim t_j$  to denote the resources occupied at time  $t_j$ ;  $t_0$  is the time of the instantiation, and  $t_n$  is the time of the termination. The resources over time are shown in

$$rs_{it} = [rs_{it_0}, \dots, rs_{it_j}, \dots, rs_{it_n}]. \quad (3)$$

Suppose the time of the first invocation is  $tv_{\text{start}}$  and the time of the last invocation is  $tv_{\text{end}}$ . Then, the relationship between the invocation time and the resource occupation time is shown in

$$t_0 < tv_{\text{start}} \ \&\& \ t_n > tv_{\text{end}}, \quad \text{if preinstantiation,} \quad (4)$$

$$t_0 \approx tv_{\text{start}} \ \&\& \ t_n \approx tv_{\text{end}}, \quad \text{if instant instantiation,} \quad (5)$$

$$t_0 \geq tv_{\text{start}} \ \&\& \ t_n \approx tv_{\text{end}}, \quad \text{if postinstantiation.} \quad (6)$$

For the preinstantiated method, this may be a very long time or even there is no invocation after the smart contract has been instantiated. It occupies resources longer than the instant instantiation. The postinstantiation has the least resource occupation time, as its instantiation time ( $t_0$ ) is not earlier than the time of the first IMR invocation, shown in (6). Then, we get the conclusion that the instant instantiation and the postinstantiation occupy the resource in less time than the predeployment method.

We define a variable  $t_s$  to measure the time saved by different instantiation methods, which is shown in (7). If its value is positive, it saves the waiting time, which is the postinstantiation case. If it is close to 0, the runtime of the smart contract instance is approximately equal to the invocation last time, which is the instant instantiation case. If its value is negative, it spends time waiting for requests, which is the predeployment case.

$$t_s = (tv_{\text{start}} - t_0) + (tv_{\text{end}} - t_n). \quad (7)$$

The total runtime saved (TS) is the summarization of all nodes which run the smart contract, shown in (8). Thus, it saves more resources in the larger blockchain networks as their node number  $n$  is more.

$$\text{TS} = \sum t_s. \quad (8)$$

### 3. Verification

In this section, we show our simulation results of different instantiation methods. As the smart contracts are required to be preinstantiated in currently available blockchains, we develop a blockchain that supports different instantiation types. It removes the requirement to prestantiate a smart contract and keeps the procedure to put the code of a smart contract to the blockchain in the predeployment stage.

The smart contract language is Java. There are two reasons. (1) The blockchain platform is developed by Java, and all nodes have the JRE (Java runtime environment) to run the Java code. Then, it saves the steps to provide an extra runtime environment for another language. (2) It is convenient to invoke smart contracts by the reflection mechanism of Java. User-defined smart contracts can be simply invoked in this way.

We provide a class as the variable to wrap the smart contract instance. It wraps behaviors that are required for different instantiation methods. The instantiation type is set in the configuration file of the smart contract. If the instantiation type is preinstantiation, its instance is created in the code putting procedure. If it is the instant instantiation, the instance is created when its creation function is called. If it is the postinstantiation, the instance is created when the specific methods are called. Those methods have calls to a platform function call which creates the instances.

```

1: Define smart contract SCA
2: ...
3: End Define
4:
5: Define smart contract SCB
6: ...
7: End Define
8:
9: Define smart contract SCC
10: //scope begin {
11: // create an instance (new process) of SCA
12: Variable sca = new Variable("SCA");
13: invoking method of sca
14: ...
15: // create an instance (new process) of SCB
16: Variable scb = new Variable("SCB");
17: invoking method of scb
18: //scope end } - sca and scb are ready to terminate (condition a), and those two smart contract instances terminate after their states
    have been confirmed in blockchain (condition b)
19: End Define
    
```

ALGORITHM 1. Smart contract invocation by a variable.

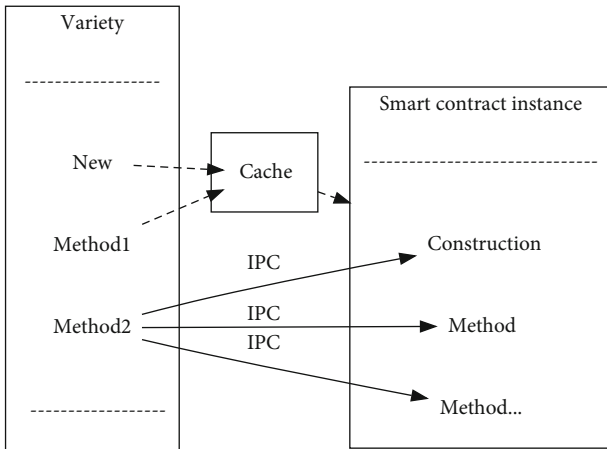


FIGURE 4: The relationship between the variable and the smart contract instance in the postinstantiation method. The instantiation of a smart contract can be delayed to the invocation of specific methods, and other methods are cached before the instantiation.

There is special handling for the termination of a variable. When a variable is out of its (definition and usage) scope, the compiler can add the corresponding code there. As we do not want to change the Java compiler, we directly put a function to terminate the smart contract instance at the end of its scope. Meanwhile, the termination function calls a function (the final checking function or FCF) which checks whether the key states of a smart contract have been sealed in the blockchain. If not sealed, FCF waits and checks. Otherwise, it exits and the smart contract destructs to release resources.

We use the file method as the interprocess communication between the process of the wrapping variable and smart contract instance as they run in different processes. The variable passes the function name and its parameters

```

1: Process(Request r)
2: ...
3: if r is IMR then
4:   create instance
5:   forward cached requests and r to instance
6: end if
7: if r is IUR then
8:   if instance is not created then
9:     cache r
10:  else
11:    forward r to instance
12:  end if
13: end if
    
```

ALGORITHM 2. Handling for IMR and IUR.

to corresponding request files, and the smart contract instance reads from those files continuously. The smart instance invokes the corresponding functions by a Java reflect mechanism.

Two kinds of verifications are carried out. (1) We perform the comparative verification between the proposed method and the preinstantiation method. It is aimed at showing the benefit of the proposed instantiation methods. (2) We also perform the verification of different instantiation methods proposed in this paper to demonstrate the advantages of different methods.

*3.1. Comparison between the Preinstantiation Method and Proposed Method.* We adopt the runtime of smart contracts to compare those two methods. The proposed instantiation is triggered by a transaction when the first request is sent from the user. In the preinstantiation method, the smart contract is preinstantiated in the predeployment stage and its instance keeps running from then on. Requests for both

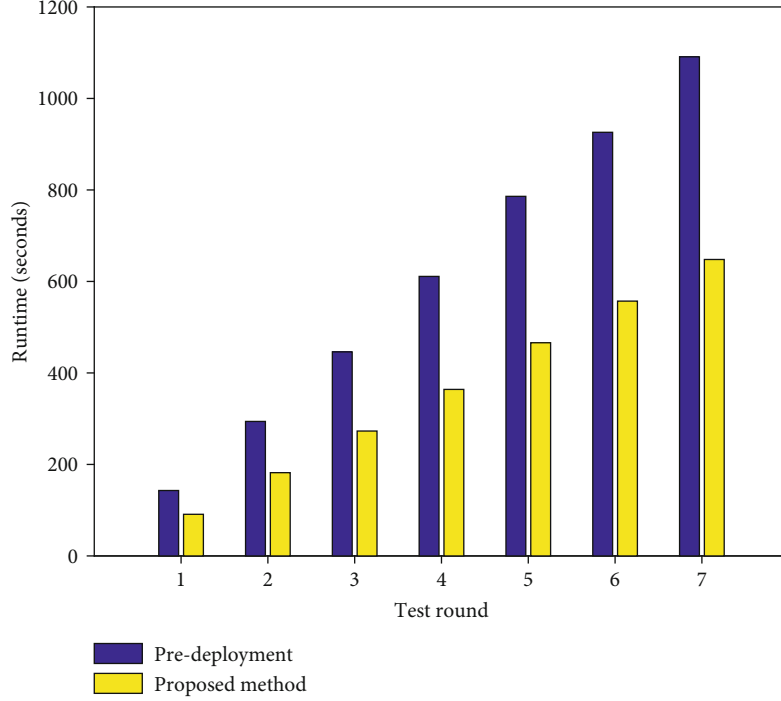


FIGURE 5: The accumulated runtime comparison between the proposed method and the preinstantiation method.

methods are sent out by a script in a random interval (the interval changes from 40 seconds to 80 seconds).

Figure 5 shows the accumulated runtime ( $t_{\text{runAcc}}$ ) of those two methods. The accumulated runtime is the summarization of the runtime of all instances, as shown in (9). In the proposed method, there may be several instances during a test round and thus  $t_{\text{runAcc}}$  is the summarization of the runtime of all instances. In the preinstantiation method, there is only one instance and  $t_{\text{runAcc}}$  is equal to the runtime of that instance. The runtime ( $t_{\text{run}^{-i}}$ ) of instance  $i$  is from the moment of its instantiation to the moment of its termination, which contains the code execution time ( $t_{\text{exec}^{-i}}$ ) and the waiting time ( $t_{\text{waiting}^{-i}}$ ), as shown in (10).

$$t_{\text{runAcc}} = \sum t_{\text{run}^{-i}} \quad (9)$$

$$t_{\text{run}^{-i}} = t_{\text{exec}^{-i}} + t_{\text{waiting}^{-i}} \quad (10)$$

From Figure 5, we see that the preinstantiation method takes more runtime than the proposed method. The reason is as follows. In (10), time  $t_{\text{waiting}^{-i}}$  of the proposed method is less than that of the preinstantiation method, because the waiting time is saved as each instance of the proposed method is terminated when not used. However, time  $t_{\text{exec}^{-i}}$  of the proposed method is more, as it instantiates a smart contract several times, while time  $t_{\text{waiting}^{-i}}$  is much bigger than time  $t_{\text{exec}^{-i}}$  in this verification.

At the first invocation, the preinstantiation method has taken 52 seconds more. This value is the interval between the first invocation and the preinstantiation. As there are intervals among invocations, the additional runtime taken by the preinstantiation increases. In fact, the runtime of the

TABLE 1: Different kinds of optimization of SCX.

Instantiation type	Postinstantiation	Local variable	Comment
A	No	No	SCY-A
B	Yes	No	SCY-B
C	No	Yes	SCY-C
D	Yes	Yes	SCY-D

preinstantiation method is the time after it has been instantiated. The instance in the proposed method is only created when there is request, and it does not occupy the interval time between different smart contract instances.

**3.2. Different Instantiation Methods.** In this section, we verify two optimizations for the proposed methods. One is the post-instantiation, and another one is to define the variable of the smart contract instance in a smaller scope (a local variable). We combine those two optimizations and get four kinds of possible instantiation optimization, shown in Table 1.

The according codes are shown in Algorithm 3. Two smart contracts are used: smart contracts SCX and SCY. SCY does different instantiations of SCX as shown in Table 1. It results in four subtypes of SCY: SCY-A, SCY-B, SCY-C, and SCY-D. SCX has two subtypes: SCXI and SCX-Post, and the latter one is the postinstantiated. And SCX provides two methods: the first one is not an instantiation must method and the second is an instantiation must method.

We have performed 28 test rounds. In those test rounds, verifications of combinations A, B, C, and D are tested in turn. The test results are shown in Figure 6. From it, we see that smart contract SCX in type A has the longest runtime,

```

1: Define smart contract SCXI
2: ...
3: End Define
4:
5: Define smart contract SCX-Post
6: ...
7: End Define
8:
9: Define smart contract SCY-A
10: do other action
11: Variable scx = new Variable("SCXI");
12: invoking method1 of scx
13: do other action
14: invoke method2 of scx;
15: do other action
16: End Define
17:
18: Define smart contract SCY-B
19: do other action
20: Variable scx = new Variable("SCX-Post");
21: invoking method1 of scx; // Method1 is cached
22: do other action
23: invoke method2 of scx; // Method2 requires to have a real SCX smart contract instance
24: do other action
25: End Define
26:
27: Define smart contract SCY-C
28: do other action
29: //scope begin {
30: Variable scx = new Variable("SCXI");
31: invoking method of scx
32: do other action
33: invoke method of scx;
34: //scope end }
35: do other action
36: End Define
37:
38: Define smart contract SCY-D
39: do other action
40: //scope begin {
41: Variable scx = new Variable("SCX-Post");
42: invoking method1 of scx; // Method1 is cached
43: do other action
44: invoke method2 of scx; // Method2 requires to have a real SCA smart contract instance
45: //scope end }
46: do other action
47: End Define

```

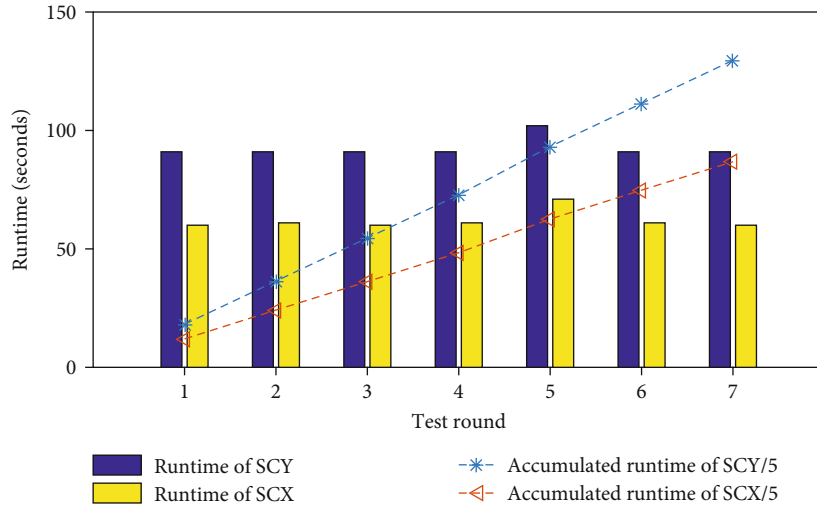
ALGORITHM 3. Different instantiation combinations.

which is shown in the first part of Figure 6. SCX is created when its according variable is created and lives until SCY-A terminates. SCX in SCY-D has the shortest runtime, which is shown in the last part of Figure 6. The reason is that SCX is created until its second method is created (the instantiation must method), and it is defined in a smaller scope, which makes it terminate early.

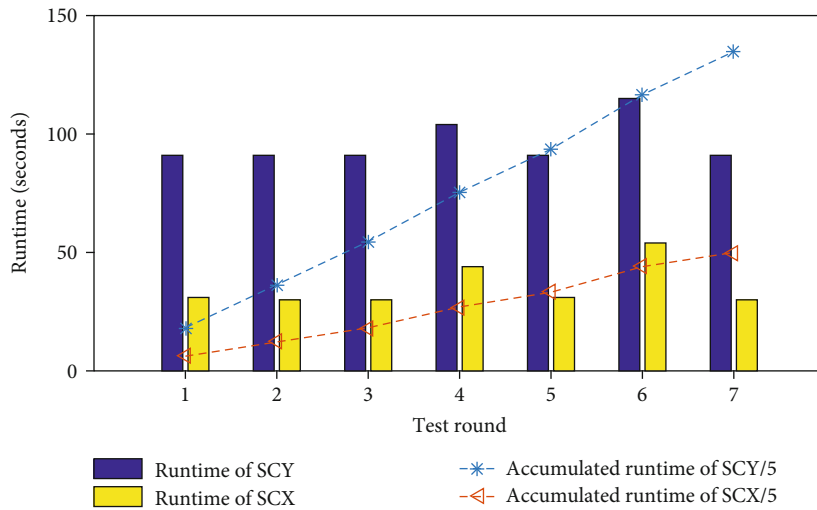
The other two methods take less time than the first one and more time than the last one. The second one delays the instantiation time of smart contract SCX, and the third one

defines smart contract SCY in a smaller scope. Then, both save the runtime.

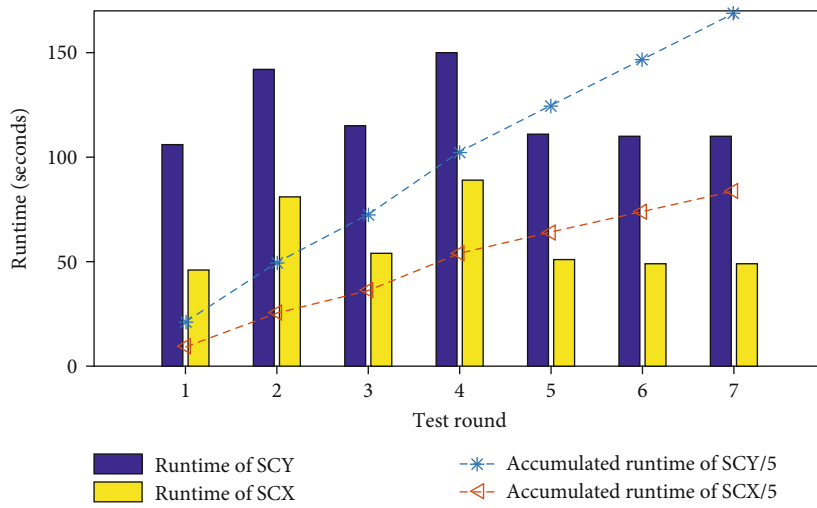
We also demonstrate the accumulated runtime in Figure 6, which is the summarization of the runtime of the instances in previous test rounds. From it, we can see that the accumulated runtime of SCX and SCY is very close in case A, which indicates that SCX lives along with SCY. On the other hand, the curve of the accumulated runtime of SCX and SCY forms a big angle in case D, which indicates that the execution time of SCX is shorter than that of SCY



(a)



(b)



(c)

FIGURE 6: Continued.



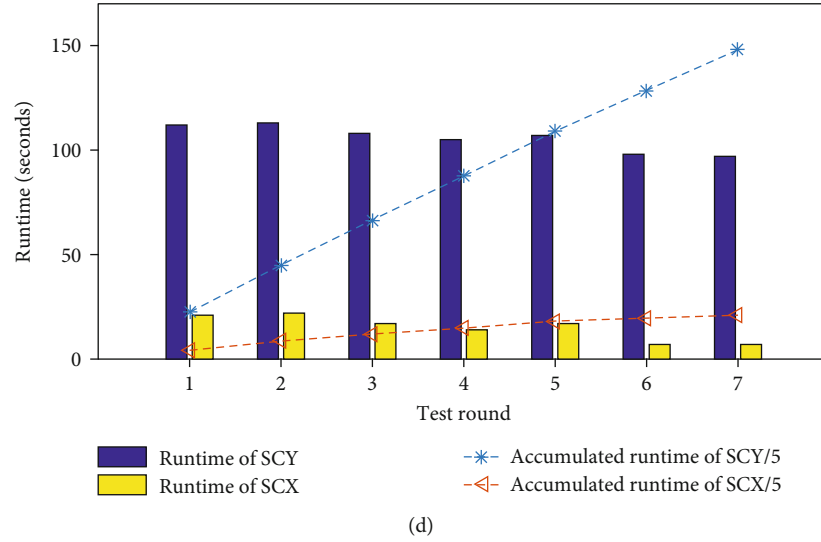


FIGURE 6: The runtime of smart contracts in different instantiation cases (A, B, C, and D from (a) to (d)). Accumulated runtime is the summarization of the runtime in the previous test run, and to make it suitable to the diagram, its value has been divided by 5.

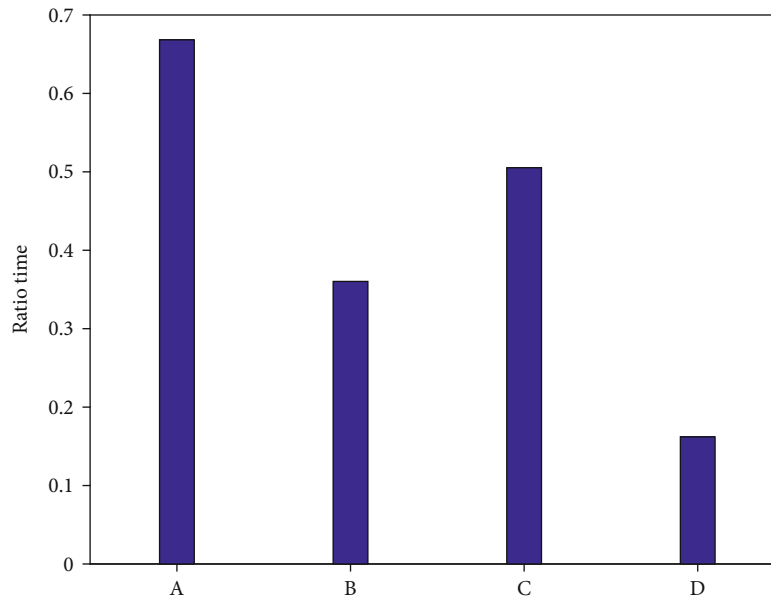


FIGURE 7: The runtime of the smart contract SCX to the runtime of smart contract SCY (ratioTime) in different instantiation cases (A, B, C, and D).

and it is optimized to save the additional resource occupation time.

As the time SCY also changes with time, and to have a more comparison of how the proportion of the runtime is saved, we use another measurement, ratioTime, which is the ratio of the runtime of the smart contract SCX to the runtime of smart contract SCY, referring to (11). The result is shown in Figure 7.

$$\text{ratioTime} = \frac{\text{runtime}(X)}{\text{runtime}(Y)}. \quad (11)$$

From Figure 7, we also see the same optimization results. In method A, SCX runs at the most time when SCY runs and has the highest ratioTime, and method D has the lowest ratioTime. The two left methods have the medium ratioTime.

#### 4. Conclusion

In this paper, we address the preinstantiation issues of smart contracts for IoT scenarios, in which blockchain nodes are composed of resource-limited IoT devices. We propose different instantiation methods to optimize the instantiation

of a smart contract, with the aim of solving the issues of the preinstantiation (it occupies computation resources even if there is no request and requires a global address). We adopt a new kind of variable to wrap the instantiation and termination of a smart contract. The smart contract instantiates when its wrapping variable is created, and it can even delay the instantiation to the moment when an instance is a must. The termination of a smart contract is performed when the variable is out of scope. At last, we perform the corresponding verifications, which show that our proposed methods occupy fewer resources (measured by the runtime). More resources can be saved if we combined different instantiation methods.

### Data Availability

No data were used to support this study.

### Conflicts of Interest

The authors declare that they have no conflicts of interest.

### Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61772352; the National Key Research and Development Project under Grant Nos. 2020YFB1711800 and 2020YFB1707900; the Science and Technology Project of Sichuan Province under Grant Nos. 2019YFG0400, 2020YFG0479, and 2020YFG0322; and the R&D Project of Chengdu City under Grant No. 2019-YF05-01790-GX.

### References

- [1] L. Hang and D. H. Kim, "Reliable task management based on a smart contract for runtime verification of sensing and actuating tasks in IoT environments," *Sensors*, vol. 20, no. 4, p. 1207, 2020.
- [2] R. Xu, Y. Chen, E. Blasch, and G. Chen, "Blendcac: a smart contract enabled decentralized capability-based access control mechanism for the IoT," *Computers*, vol. 7, no. 3, p. 39, 2018.
- [3] S. Huh, S. Cho, and S. Kim, "Managing IoT devices using blockchain platform," in *2017 19th international conference on advanced communication technology (ICACT)*, pp. 464–467, Bongpyeong, South Korea, 2017.
- [4] H. Cheng, Y. Chen, N. Xiong, and F. Li, "Layer-based data aggregation and performance analysis in wireless sensor networks," *Journal of Applied Mathematics*, vol. 2013, 12 pages, 2013.
- [5] Z. Guan, G. Si, X. Zhang et al., "Privacy-preserving and efficient aggregation based on blockchain for power grid communications in smart communities," *IEEE Communications Magazine*, vol. 56, no. 7, pp. 82–88, 2018.
- [6] Q. Zhang, C. Zhou, N. Xiong, Y. Qin, X. Li, and S. Huang, "Multimodel-based incident prediction and risk assessment in dynamic cybersecurity protection for industrial control systems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 46, no. 10, pp. 1429–1444, 2015.
- [7] K. Huang, Q. Zhang, C. Zhou, N. Xiong, and Y. Qin, "An efficient intrusion detection approach for visual sensor networks based on traffic pattern learning," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 47, no. 10, pp. 2704–2713, 2017.
- [8] O. Alkadi, N. Moustafa, B. Turnbull, and K. R. Choo, "A deep blockchain framework-enabled collaborative intrusion detection for protecting IoT and cloud networks," *IEEE Internet of Things Journal*, 2020.
- [9] H. Su, B. Guo, Y. Shen, T. Li, C. Qing, and Z. Zhang, "A solution for state conflicts of smart contract in interaction with non-blockchain," in *IEEE INFOCOM 2020-IEEE conference on computer communications workshops (INFOCOM WKSHPS)*, pp. 382–387, Toronto, ON, Canada, Canada, 2020.
- [10] G. Caldarelli, "Understanding the blockchain oracle problem: a call for action," *Information*, vol. 11, no. 11, p. 509, 2020.
- [11] G. Destefanis, M. Marchesi, M. Ortu, R. Tonelli, A. Bracciali, and R. Hierons, "Smart contracts vulnerabilities: a call for blockchain software engineering?," in *2018 international workshop on Blockchain oriented software engineering (IWBOSE)*, pp. 19–25, Campobasso, Italy, 2018.
- [12] Y. Liu, Q. Lu, X. Xu, L. Zhu, and H. Yao, "Applying design patterns in smart contracts," in *International Conference on Blockchain*, pp. 92–106, Cham, 2018.
- [13] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pp. 442–446, Klagenfurt, Austria, 2017.
- [14] A. Reyna, "On blockchain and its integration with IoT. Challenges and opportunities," *Future Generation Computer Systems*, vol. 88, pp. 173–190, 2018.
- [15] W. Wu, N. Xiong, and C. Wu, "Improved clustering algorithm based on energy consumption in wireless sensor networks," *IET Networks*, vol. 6, no. 3, pp. 47–53, 2017.
- [16] I. Kuzminykh, A. Carlsson, M. Yevdokymenko, and V. Sokolov, "Investigation of the IoT device lifetime with secure data transmission," in *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, pp. 16–27, Springer, Cham, 2019.
- [17] A. Bahga and V. Madiseti, "Blockchain platform for industrial Internet of things," *Journal of Software Engineering and Applications*, vol. 9, no. 10, pp. 533–546, 2016.
- [18] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Generation Computer Systems*, vol. 107, 2017.
- [19] M. Knecht and B. Stiller, "SmartDEMAP: a smart contract deployment and management platform," in *IFIP international conference on autonomous infrastructure, management and security*, pp. 159–164, Cham, 2017.
- [20] C. F. Liao, C. J. Cheng, K. Chen, C. H. Lai, T. Chiu, and C. Wu-Lee, "Toward a service platform for developing smart contracts on blockchain in BDD and TDD styles," in *2017 IEEE 10th Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 133–140, Kanazawa, Japan, 2017.
- [21] C. Sillaber and B. Waltl, "Life cycle of smart contracts in blockchain ecosystems," *Datenschutz Und Datensicherheit Dsd*, vol. 41, no. 8, pp. 497–500, 2017.