IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 12 March 2024, accepted 1 April 2024, date of publication 10 April 2024, date of current version 26 April 2024.

*Digital Object Identifier 10.1109/ACCESS.2024.3387300*

RESEARCH ARTICLE

# MECAT: Memory-Safe Smart Contracts in ARM TrustZone

**SEONGHWAN PARK[1], HAYOUNG KANG[1], SANGHUN HAN[1], JONGHEE M. YOUN[2], AND DONGHYUN KWON[3]**

[1]Department of Information Convergence Engineering, Pusan National University, Geumjeong-gu, Busan 46241, Republic of Korea
[2]Department of Computer Engineering, Yeungnam University, Gyeongsan 36461, Republic of Korea
[3]School of Computer Science and Engineering, Pusan National University, Geumjeong-gu, Busan 46241, Republic of Korea

Corresponding authors: Jonghee M. Youn (youn@yu.ac.kr) and Donghyun Kwon (kwondh@pusan.ac.kr)

This work was supported in part by the Ministry of Science and Information and Communication Technology (MSIT), South Korea, under the Special Research and Development Zone Development Project (Research and Development)-Development of Research and Development Innovation Valley Support Program supervised by the Innovation Foundation under Grant 2023-DD-RD-0152; and in part by MSIT, under the Information Technology Research Center (ITRC) Support Program supervised by the Institute for Information and Communications Technology Planning and Evaluation (IITP) under Grant IITP-2024-RS-2023-00259967.

**ABSTRACT** A smart contract is a program executed on a blockchain. However, once the smart contract is deployed on the blockchain, it becomes visible to all participants and remains immutable. Thus, any sensitive information or vulnerabilities in smart contracts can be exposed to potential attackers. To protect the confidentiality of smart contracts, existing studies execute smart contracts in a trusted execution environment (TEE). However, they still suffer from vulnerabilities in smart contracts and potential memory-vulnerability problems. If an attack such as privilege escalation occurs by exploiting this vulnerability, the TEE can have a detrimental effect on the entire system as it has the most privileges in the system. To mitigate the memory vulnerability of the smart contracts in TEE, we propose MECAT, a prototype for memory-safe confidential smart contracts. In essence, MECAT runs smart contracts written in Rust, a memory-safe language, in ARM TrustZone. And MECAT is developed as a software library, allowing developers to easily apply MECAT to their smart contracts. According to our evaluation, MECAT only incurs a 1.36x performance overhead and 0.30% power overhead in single-node environments and can process the 16 clients concurrently in the network made with 8 peer nodes.

**INDEX TERMS** Blockchain, smart contracts, hyperledger fabric, rust, memory-safety, ARM TrustZone.

## I. INTRODUCTION

A smart contract is a program executed on a blockchain when specific predefined conditions are met. Notably, it facilitates trustless and decentralized transactions by eliminating the need for intermediaries and enabling programmable agreements across various domains such as finance, supply chain, smart marketplaces, and decentralized applications (DApps) [1], [2], [3]. However, once a smart contract is deployed on the blockchain, it becomes visible to all participants and remains immutable, posing inherent security challenges. Specifically, the lack of confidentiality about application code and execution states can expose vulnerabilities in the smart contract to potential attackers. Even worse, since the contract code cannot be altered after registration on the blockchain, patching vulnerabilities in the deployed smart contracts is challenging. Consequently, financial losses and other adverse consequences result if attackers exploit vulnerable smart contracts through carefully crafted payloads.

To mitigate these security problems, some researchers have leveraged Trusted Execution Environments (TEE) to provide confidential smart contracts [4]. TEE is a security feature that ensures the confidential execution of sensitive operations, protecting them from unauthorized access even in the presence of a compromised overall system.

Thanks to distributed systems, blockchain technology can provide several benefits in IoT networks comprised of

The associate editor coordinating the review of this manuscript and approving it for publication was Kashif Sharif.

embedded devices, such as security, autonomy, resilience, and anonymity [5], [6], [7]. Furthermore, since blockchain technology is widely adopted for IoT, embedded devices are rising as an important smart contract carrier [8], [9]. Most embedded devices are equipped with ARM chips. ARM provides a TEE technique called ARM TrustZone (TrustZone) [10]. Because of these points, some researchers worked on providing confidential smart contracts using TrustZone [11], [12].

Especially, TZ4Fabric [12] ensures the confidentiality of smart contracts and execution states in Hyperledger Fabric [13], by executing smart contracts within TrustZone. Although the above work successfully provides confidentiality about smart contracts using TEE, it introduces potential risks because of bloating the code size of TEE. Moreover, previous work [12] uses memory-unsafe languages such as C/C++. However the TrustZone provides a single secure OS in TEE, when the privilege escalation attack occurs to an application within TEE, it can have a detrimental impact not only on secure OS but also on the other applications [14].

In this paper, we propose MECAT, a technique to provide memory-safe and confidential smart contracts. Specifically, MECAT adopts TrustZone to protect the confidentiality of smart contracts. And, to mitigate memory vulnerabilities of confidential smart contracts, MECAT uses memory-safe programming language Rust [15] to write smart contracts. Notably, MECAT is provided as a software library, allowing developers to easily enjoy memory-safe confidential smart contracts by applying MECAT APIs to their smart contracts.

In summary, MECAT has three main contributions as follows.

- We mitigate the impact of memory vulnerability problems caused by confidential smart contracts executed in TEE.
- We provide an easy-to-use and compatible design with a software library and modularized components.
- We show the impact of MECAT on the network and hardware by prototyped implementation with Hyperledger Fabric and Raspberry Pi 3B+.

## II. BACKGROUND AND MOTIVATION

### A. HYPERLEDGER FABRIC AND CHAINCODE

Hyperledger Fabric (Fabric) [13] is an open-source blockchain platform hosted by the Linux Foundation. Fabric is a permissioned blockchain system; it runs distributed applications written in general-purpose programming languages, such as Java, Go, and JavaScript. Distributed application, i.e., smart contract., is called chaincode in Fabric.

Chaincode is constructed of handlers for handling client input and business logic that processes the transaction. When the client invokes chaincode, the handler will receive control of the transaction and call the appropriate business logic for processing the transaction. The complexity and size of the chaincode are determined by business logic.

Algorithm 1 shows the pseudo-code of the simple chaincode handler, which keeps the name of the user and value in a ledger. For this, it has three subcommands, i.e., Create, Query, and Add, each creates a user with an initialized value, queries the value of the user, and adds the value of the user. Handler has user inputs as parameters. When the client invokes a chaincode, handler has the control of the transaction first. After that, it parses the function type and arguments from client input. (line 2) Finally, the handler calls the appropriate function to process the business logic as intended (lines 3, 5, and 7) and return the result value to the client (lines 4, 6, and 8).

**Algorithm 1** Pseudo-code of the chaincode handler

```
   Input:
   Client input parameters CI
1  function handler(CI)
2      function, args = getFunctionAndParameters(CI)
3      if function == Create then
4          return create(args)
5      else if function == Query then
6          return query(args)
7      else if function == Add then
8          return add(args)
9      else
10         return Error
11     end
12 end
```

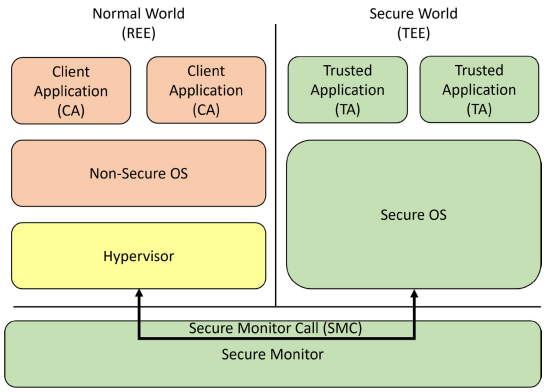


**FIGURE 1.** TrustZone architecture overview. The green boxes mean trusted areas. The world switching has invoked with SMC through the secure monitor.

### B. ARM-TRUSTZONE

TrustZone is a hardware security extension designed to establish secure execution environments [10], [16]. TrustZone introduces two distinct domains, namely normal world and secure world. The secure world has higher privileges in comparison to the normal world, thereby guaranteeing that the resources of the secure world remain inaccessible to the normal world. The identification of the world state is determined by the value of non-secure (NS) bit in the secure configuration register.

```
01  let r;                  // -------+ lifetime 'a
02      {                   //        |
03          let x = 5;      // -+ lifetime 'b
04          r = &x;         //  | r referenced x
05      }                   // -+       |
06  println!("r: {}", r);   // --------+  dangling reference
```

**FIGURE 2.** Dangling pointer prevention in the Rust with lifetime.

The architecture of TrustZone is depicted in Figure 1. The normal world is constructed with client applications (CAs), a non-secure OS, and a hypervisor, and the secure world is constructed with a secure OS and trusted applications (TAs). The secure monitor has the highest privilege and is responsible for switching between the normal world and the secure world. TrustZone introduced a special instruction known as Secure Monitor Call (SMC) for world switching between the normal world and the secure world. When the SMC occurs within the normal world, the processor enters monitor mode. In monitor mode, the secure monitor stores the context of the normal world and flips the NS bit. Afterward, it restores the context of the secure world, and vice versa.

### C. RUST

Rust [15] is a programming language that has been designed for high performance and safety. Rust offers memory safety by implementing ownership, borrowing, and lifetimes. Rust has a fundamental principle that dictates all values can only have a single owner at any given time. When the owner of the object is no longer within the scope, the value will be dropped. In the context of the Rust, references can be created by borrowing. Typically, the borrowed value demonstrates immutability. For mutating the referenced value, a mutable type of reference is needed. However, it is important to note that the value can only be associated with a single reference at a moment when it is referred to as a mutable type.

The lifetimes in Rust serve the purpose of preventing dangling references in Rust. Each value has a lifetime at the scope in which it is declared, and the reference of this value becomes invalid once it goes beyond this scope. Figure 2 shows the lifetime exception example of the Rust. The variable r is declared within the scope of lifetime a on line 01, and it references value x that is declared within the scope of lifetime 'b on line 04. In line 06, variable r uses the referenced value x after the lifetime of b has ended. Rust compilers can detect these errors at compile time, allowing developers to easily rectify the code with the assistance of error messages.

### D. CONFIDENTIAL SMART CONTRACTS

Since the application code and execution state of the smart contracts are exposed transparently in the blockchain network, confidential smart contracts have been researched to address this problem.

Hawk [17] provides confidentiality to smart contracts, using zero-knowledge proofs. To achieve this, smart contracts are off-chain; only zero-knowledge proofs are recorded in the ledger. However, the zero-knowledge proof incurs a relatively high-performance overhead for TEE-assisted confidential smart contracts.

Intel-SGX (SGX) [18] is a hardware security extension that provides a TEE in Intel processors. Ekiden [19], ShadowEth [20], CCF [21], CONFIDE [22], FPC [23], and Phala [24] utilize SGX to execute confidential smart contracts. Especially CCF [21], CONFIDE [22], FPC [23], and Phala [24] are integrated into blockchain platforms for practical use. However, implementing these solutions directly in TrustZone is challenging because of the lack of hardware support, such as remote attestations, and the relatively limited memory resources compared with SGX.

TSC-VEE [11], and Tz4Fabric [12] studied confidential smart contracts in TrustZone. TSC-VEE [11] has implemented the Solidity VM into the TEE to enable confidential smart contracts. It provides the instruction set for the Solidity bytecode. Furthermore, memory management and prefetching technologies were introduced to overcome memory limitations and enhance performance. However, it requires additional physical memory space for the VM. This can hinder the performance of embedded devices because it reduces the available memory resources. Tz4Fabric [12] provides confidential smart contracts on the TrustZone at the Hyperledger Fabric platform [13]. Since Fabric supports Go, Node.js, and Java for writing smart contracts, it is necessary to rewrite the legacy contracts in a memory-unsafe programming language, i.e., C/C++. Because smart contracts can be written by third-party developers, memory-unsafe smart contracts can cause potential security risks in the TEE. MECAT adopts the memory-safe programming language Rust to write memory-safe smart contracts. This approach can address the potential security risks associated with third-party smart contracts.

### E. SECURITY OF TRUSTZONE

Although TrustZone provides TEE to construct a reliable system, it has poor applicability because of the absence of robust TEE runtime or due to architectural flaws [10], [25]. To address this problem vTZ [26], TrustShadow [27], PrOS [28], Sanctuary [29], and RusTEE [30] have been studied.

vTZ [26] provides virtualization to TrustZone with a software-only design. This study virtualized the TEE as a VM, executed it in the REE, and guaranteed isolation between them. However, because of the limitations of the software-only design, it incurs a high-performance overhead.

TrustShadow [27], PrOS [28], and Sanctuary [29] use a hardware feature TrustZone Address Spaces Controller to achieve physical memory isolation between TAs. However, this design can cause significant overhead due to world switching caused by memory mapping, and the absence of the TrustZone Address Spaces Controller can hinder the deployment of the confidential smart contract.

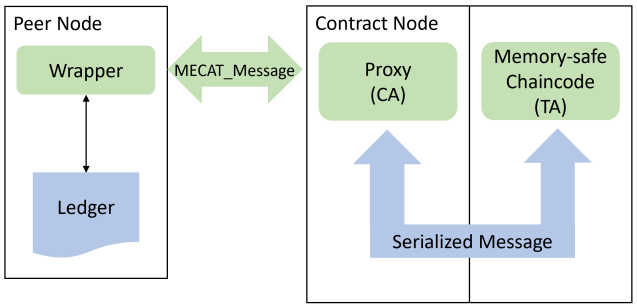


**FIGURE 3.** MECAT architecture overview. The peer node is run on the host machine and the contract node is run on the embedded device, that supports TrustZone.

RusTEE [30] provides a Rust-based TA SDK for developing memory-safe TAs. This design can address the potential risks from memory-unsafe TAs and enhance TEE security. However, the Rust-based TA SDK does not support smart contract development. The design of MECAT can complement this problem by providing confidential and memory-safe smart contracts.

## III. THREAT MODEL

We assume the fundamental components of the secure world are trusted, such as the boot loader, firmware, security monitor, and secure OS. Meanwhile, the components of the normal world, such as non-secure OS, and user space, are untrusted. This is a common scenario recognized in previous studies that adopt TrustZone [11], [12], [30].

The blockchain network is permission-based, so most of the participants are benign, and the smart contracts are not malicious but can contain memory vulnerabilities. The input provided by the user through the client can cause memory corruption bugs in smart contracts. Attacks against TrustZone itself or the network, such as side-channel attacks [31], [32], and DoS [33] are out of scope.

## IV. MECAT OVERVIEW

In MECAT, we provide confidentiality about code and execution state to the smart contract by adopting TrustZone, and mitigate the memory vulnerabilities of the smart contract by using Rust. The ideal solution for the confidential smart contract is to execute the whole node in the embedded device with TEE. However, it has challenges due to the limited persistent memory resources and computing power of embedded devices [5]. In the case of persistent memory, the blockchain network keeps all transactions made in the network as a ledger, because of this, the size of the ledger will increase larger than the persistent memory of devices. In the case of computing power, the consensus algorithms require high computing power. Furthermore, when the whole node runs in the TEE, it can bring large attack surfaces because of its code size.

To address these challenges, we partitioned the smart contract into three main components, *wrapper*, *proxy*, and *memory-safe chaincode*. Figure 3 shows the architecture of the MECAT. The peer node is run on a host machine that includes the *wrapper* and the ledger and is responsible for consensus. Since the peer node runs on a host machine, we can address the limitation of persistent memory resources and computation power. The *wrapper* is installed, instantiated, and deployed as a smart contract in the blockchain network. Since our design does not affect parts other than transaction execution, MECAT can perform transactions in the *memory-safe chaincode* by calling the *wrapper* with a standard approach.

```
01 MECAT_Message
02 {
03     Context: C,
04     Subcommand: S,
05     Parameter: P,
06     Message: M
07 }
```

**FIGURE 4.** The structure of the *MECAT_Message*. This includes information about the context state, subcommand, parameter, and message.

The contract node supports TrustZone and includes the *proxy* and the *memory-safe chaincode*. The *proxy* is run in the normal world of the contract node and establishes communication between the *wrapper* and *memory-safe chaincode*. The *memory-safe chaincode* is run in the secure world of the contract node. Because of this, we can provide confidentiality to smart contracts. In the contract node, all components are implemented with the memory-safe programming language Rust, which can guarantee robustness against memory vulnerabilities, thereby mitigating the potential security risks in the secure world.

From this design, we have overcome the constraint of limited memory resources and computing power of embedded devices. In addition, we minimized the code size and attack surface within the TEE by executing only the business logic of the smart contracts within the TEE.

## V. TRANSACTION FLOW

The peer and contract nodes communicate with each other using *MECAT_Message*, which contains the transaction information. Figure 4 shows the structure of the *MECAT_Message* consists of four fields; context, subcommand, transaction parameter, and message. The context field is one of the following seven contexts, showing the current state.

- `InvokeRequest` means initiating the transaction to the *memory-safe chaincode*.
- `GetRequest` means *memory-safe chaincode* requested getting information from the ledger.
- `GetResponse` means *proxy* handled the request from *memory-safe chaincode*.
- `SetRequest` means *memory-safe chaincode* requested to update the ledger to *proxy*.

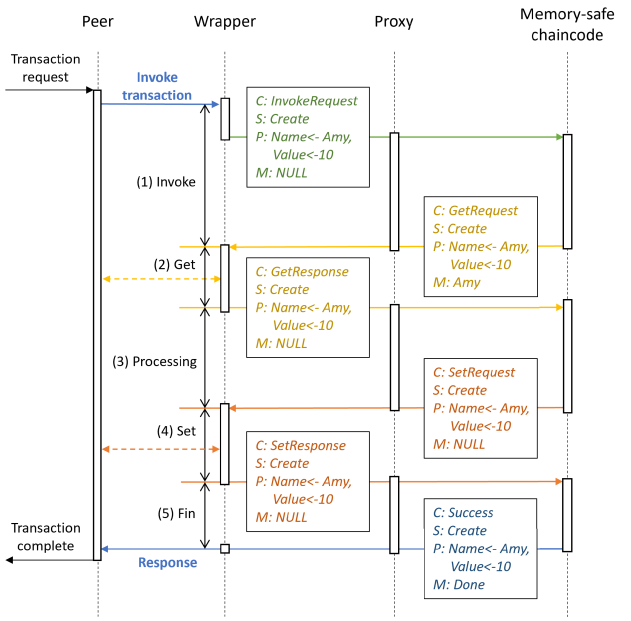


**FIGURE 5.** Create transaction flow in MECAT, solid arrows and square boxes depict message transfer between *memory-safe chaincode* and *wrapper* and dashed lines depict ledger access between *wrapper* and peer.

- `SetResponse` means *proxy* handled the request from *memory-safe chaincode*.
- `Success` means the transaction was successfully processed.
- `Failed` means an error occurred during the transaction processing.

The subcommand and parameter fields are parsed from the client transaction request input. The message field contains the request parameter and results of the ledger access request.

In Figure 5, we illustrate the transaction processing flow of the MECAT with a specific scenario, a client invokes a transaction to register a new user Amy with an initial value of 10 with the subcommand Create. The square boxes display the contents of *MECAT_Message* for each phase. (1) Invoke phase *wrapper* creates a *MECAT_Message* with `InvokeRequest` and the client input parameters (Create, Amy, 10) and then sends it to *memory-safe chaincode* via *proxy*. When *memory-safe chaincode* receives the *MECAT_Message*, it responds with `GetRequest` to request ledger access. (2) Get phase *wrapper* receives ledger access request with `GetRequest` state, it retrieves the ledger with an argument parsed from the message field. Then, *wrapper* sends `GetResponse` with the result of the ledger access. Since the user Amy is not registered in the ledger, the message field will be filled with the result, NULL. (3) Processing phase *memory-safe chaincode* receives the result of (2), it determines whether to update the ledger according to the result contained in the message filed. After that, sends `SetRequest` to *wrapper* for a request to update the ledger for registering user Amy to the network. (4) Set phase *wrapper* receives the `SetRequest`, updates the ledger, and then replies with `SetResponse`. (5) Fin phase *memory-safe chaincode* receives `SetResponse`, it replies the `Success` to with a message "Done" to *wrapper* for terminating the transaction.

In this flow, *wrapper* can handle ledger access requests from *memory-safe chaincode* with context state and message field. Therefore, the *memory-safe chaincode* can access the ledger when needed.

## VI. COMPONENTS

In this section, we elaborate on the three main components, i.e., *wrapper*, *proxy*, *memory-safe chaincode*, and APIs.

### A. WRAPPER

The *wrapper* is deployed as a smart contract in the blockchain network. Due to this, a peer can invoke a transaction through the *wrapper*. During the transaction processing, it is responsible for processing ledger access requests from the *memory-safe chaincode*. When the *wrapper* receives a transaction request from the peer, *wrapper* converts the request into *MECAT_Message* and sends it to the *proxy*.

**Algorithm 2** Pseudo-code of the *wrapper*

```
   Input:
   Client input parameters CI
   MECAT_Message received from proxy MFP
1  function wrapper(CI)
2      invokeHandler(CI)
3      while do
4          MFP = receiveFromProxy()
5          if MFP.C == GetRequest then
6              getRequestHandler(MFP)
7          else if MFP.C == SetRequest then
8              setRequestHandler(MFP)
9          else if MFP.C == Success then
10             return transactionEnd(MFP)
11         else
12             return contextError(MFP)
13         end
14     end
15 end
```

Algorithm 2 shows the pseudo-code of the *wrapper*. First, when the *wrapper* receives a transaction process request from the peer, *wrapper* calls the `invokeHandler` function. In the `invokeHandler` function, it creates a *MECAT_Message* with `InvokeRequest` and sends it to the *proxy*. After the `InvokeRequest`, *wrapper* receives the response from *proxy* and processes the request depending on the context until the response with `Success` (lines 3 - 14). When the message which has `GetRequest` context arrives, *wrapper* gets the information about parameters from the ledger and replies as the `GetResponse` context by calling the `getRequestHandler` (lines 5 - 6). Otherwise, when the *memory-safe chaincode* responds as `SetRequest`

context, the *wrapper* updates the ledger with received parameters and replies with the `SetResponse` context (lines 7 - 8). If the `Success` has arrived, *wrapper* returns the result to the peer and ends the transaction (lines 9 - 10). If the context of the *MECAT_Message* is invalid, the *wrapper* shows the error message and halts the *wrapper* processor (lines 11 - 12).

In this design, we separated the ledger access functions from the transaction processor to the *wrapper*, which can provide compatibility and adaptability.

### B. PROXY

The *proxy* establishes an interface between the *wrapper* and the *memory-safe chaincode*. When the *proxy* receives a *MECAT_Message* as a byte stream from the *wrapper*, it reconstructs the *MECAT_Message* as an object and then forwards it to *memory-safe chaincode* for processing the transaction.

**Algorithm 3** Pseudo-code of the *proxy*

```
   Input:
   MECAT_Message received from wrapper MFW
   MECAT_Message sent to wrapper MTW
1  function proxy()
2  |  MFW = receiveFromWrapper()
3  |  MTW = invokeContract(MMF)
4  |  sendToWrapper(MTW)
5  end
```

Algorithm 3 shows the pseudo-code of the *proxy*. First, *proxy* calls the `receiveFromWrapper` to receive a *MECAT_Message* as a byte stream from *wrapper* and reconstruct the object (line 2). Then, invoke the *memory-safe chaincode* with `invokeContract` function and *memory-safe chaincode* returns *MECAT_Message* as a return value (line 3). The `invokeContract` function serializes the *MECAT_Message* as a vector and deserializes it to the *MECAT_Message* when the result is returned. As a last, in `sendToWrapper` sends the returned *MECAT_Message* as a byte stream to *wrapper* (line 4).

Since the *proxy* has limited capabilities, i.e., only establishing an interface between the *wrapper* and the *memory-safe chaincode*, it provides high versatility. It is independent of *memory-safe chaincode* and *wrapper*.

### C. MEMORY-SAFE CHAINCODE

The *memory-safe chaincode* is executed as a TA in the secure world and processes the transaction received from *wrapper* following the context. Since *memory-safe chaincode* is written in Rust and runs in the TEE, it can guarantee confidentiality and memory safety. *memory-safe chaincode* is constructed similarly to Algorithm 1 except invoked by *wrapper* and delegate the ledger access to *wrapper* via *MECAT_Message*.

**Algorithm 4** Pseudo-code of the handler part of *memory-safe chaincode*

```
   Input:
   MECAT_Message received from proxy MFP
   MECAT_Message returned to proxy MTP
1  function memorySafeChaincode()
2  |  MFP = invokeFromProxy()
3  |  MTP = invokeHandler(MFP)
4  |  return MTP
5  end
6
7  function invokeHandler(MFP)
8  |  subcommand = getSubcommand(MFP)
9  |  if subcommand == Create then
10 |  |  return createHandler(MFP)
11 |  else if subcommand == Query then
12 |  |  return queryHandler(MFP)
13 |  else if subcommand == Add then
14 |  |  return addHandler(MFP)
15 |  else
16 |  |  return error(Error)
17 |  end
18 end
```

Algorithm 4 shows the pseudo-code of the handler part of the *memory-safe chaincode*, which receives a transaction request from the *wrapper* and calls the `invokeHandler`(lines 1-5). After that, the `invokeHandler` function transfers the processing flow as follows according to the subcommand (lines 7-18).

We provide *MECAT_Message* creation functions which require arguments *MECAT_Message* structure and a string for the message field. Algorithm 5 shows the pseudo-code of subcommand Create as shown in Figure 5. When flow riches as a subcommand handler, it gets the context state from *MECAT_Message* by calling the function `getContext` (line 2). After this, *memory-safe chaincode* handling the transaction follows as context state. In `InvokeRequest` state, *memory-safe chaincode* returns a `GetRequest` message with the name of the user which will be created by calling the `createGetRequest` function (lines 3 - 5). In `GetResponse` state, *memory-safe chaincode* checks the user already exists if the user does not exist, it responds with `SetRequest` (lines 6 - 9). If the user already exists, it responds with a `Failed` (lines 10 - 12). In `SetResponse` state, *memory-safe chaincode* checks the result of the ledger update request. If the update is completed successfully, *memory-safe chaincode* responds with `Success` (lines 13 - 16). If an error has occurred during the update, *memory-safe chaincode* responds with `Failed` and an error message (lines 17 - 19). When the context has no defined state, *memory-safe chaincode* response with `Failed` and an error message (lines 20 - 22).

**Algorithm 5** Pseudo-code of the create handler of *memory-safe chaincode*

```
Input:
MECAT_Message received from proxy MFP
function createHandler(MFP)
    context = getContext(MFP)
    if context == InvokeRequest then
        message = getPayload(MFP).name
        return createGetRequest(MFP, message)
    else if context == GetResponse then
        message = getMessage(MFP)
        if message == NULL then
            return createSetRequest(MFP, NULL)
        else
            return createFailed(MFP, Error)
        end
    else if context == SetResponse then
        message = getMessage(MFP)
        if message == NULL then
            return createSuccess(MFP, Done)
        else
            return createFailed(MFP, Error)
        end
    else
        return createFailed(MFP, Error)
    end
end
```

## VII. EVALUATION

### A. ENVIRONMENT SETTING

We implemented MECAT on a Raspberry Pi 3B+ with OP-TEE (version 3.18) a popular open-source secure OS with support for TrustZone. To develop memory-safe smart contracts, we use the OP-TEE Rust SDK [34] which is based on RusTEE [30]. The *wrapper* is written in Go, which is a Fabric chaincode support language.

We evaluate the performance of MECAT, on Hyeprledger Fabric blockchain network. We built a blockchain network using Hyperledger Fabric (version 1.4) [13]. The peer node and *wrapper* are run on the server machine with an i9-10900k processor clocked at 3.7 GHz, 64 GB of RAM, and Ubuntu 20.04. The *wrapper* and the peer node are run as a docker container.

We used Hyperledger Caliper [35] (version 0.3.2) as a benchmark tool to evaluate the performance impact of MECAT on the network. For evaluation, we implemented MECAT with a simple smart contract, that creates a user, queries the value, and adds the value of the user. Then compare this with three cases, Native, REE, and MECAT. In Native, the smart contract is written in GO and executed as a Docker container on the same host machine as the peer node. In REE, the smart contract is written in Rust and executed in the embedded device, similar to MECAT, the only difference is business logic is not executed in the secure world.

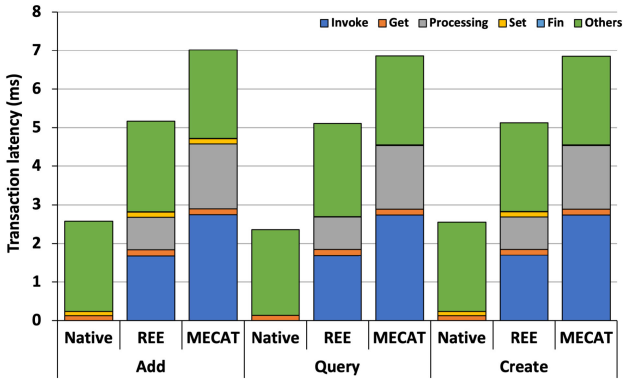


**FIGURE 6.** Transaction latency according to subcommand, including processing time.

### B. NODE EVALUATION

We evaluated MECAT in a single peer node environment to measure the overhead of transaction processing and power, for this, we executed 100 transactions in 10 seconds.

#### 1) TRANSACTION LATENCY

We measured the transaction latency at the peer about each subcommand. For this, we executed each subcommand with ten transactions per second for ten seconds and took the mean of the results.

Figure 6 shows the latency of Native, REE, and MECAT. The REE and MECAT show 2.14x-2.32x and 2.91x-3.38x of latency overhead compared with Native. However, when compared REE and MECAT show 1.36x-1.46x of execution time overhead. The subcommand Query shows less execution time overhead than other subcommands. Because the Query does not need to update the ledger does not include the Set phase. Subcommands Create and Add show mostly the same execution time overhead since they have almost identical transaction flow.

#### 2) PERFORMANCE ANALYSIS

To assess the influence of MECAT on transaction latency, we measured the execution time of each subcommand along with its respective phase. To measure the execution time of phases, we utilized the Go time package within *wrapper*.

Figure 6 shows the execution time followed by phase. Others indicate time beyond transaction processing, encompassing the period when the peer receives the client's request, invokes the smart contract, and returns the result to the client. Since the design of MECAT does not affect on blockchain network except for smart contract executions, Others have a nearly similar execution time. In addition, the Get and Set phases i.e., ledger access time, show nearly similar execution time as Others, because the ledger access is processed in the host machine. The Invoke and Processing phases include *MECAT_Message* transfer time and execution time within the contact node, these show significant overhead over the Native. Furthermore, in MECAT, the Invoke and Processing

**TABLE 1. The micro evaluation result of MECAT including message serial/deserialization time and world switching.**

| | |
|---|---|
| serialization | 11.48 MB/s |
| deserialization | 3.00 MB/s |
| world switching | 0.30 ms |

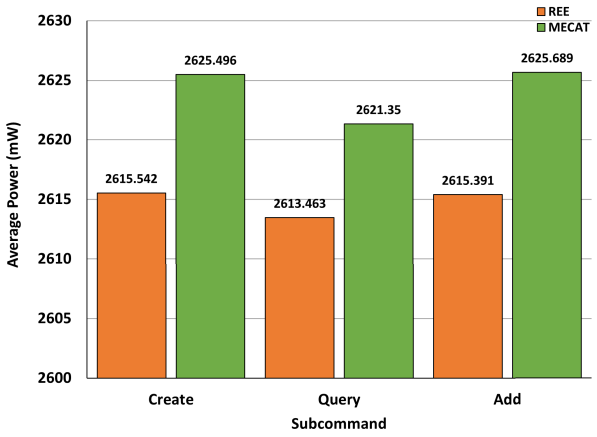


**FIGURE 7. Power consumption of each subcommand in 10 transactions per second rate.**

phases include the *MECAT_Message* serial/deserialization and world switching time between normal and secure worlds, they show more overhead than REE.

Table 1 shows the micro-benchmark result of MECAT. Since the size of *MECAT_Message* is not more than a hundred bytes, the serial/deserialization time is trivial. We measure the world switching time by invoking the TA in CA without parameters and immediately returning from the secure world to the normal world. The world switching time takes the most of performance overhead between REE and MECAT.

#### 3) POWER MEASUREMENT

We measured the power consumption of MECAT and REE using the Monsoon power monitor. We chose not to compare MECAT with Native due to the significant electric power consumption of Native, which runs on the host machine.

Figure 7 shows the average power consumption during the processing of 100 transactions over 10 seconds. Although MECAT requires six world switchings during transaction processing, the power consumption associated with world switching is remarkably small [36]. Since the only difference between MECAT and REE is processing the transactions in secure world, the average power shows a negligible overhead of 0.30%-0.39%. Therefore, the increase in power consumption due to world switching is considered negligible. Consequently, MECAT offers a memory-safe smart contract solution with high energy efficiency.

### C. NETWORK EVALUATION

To measure the impact of MECAT in the blockchain network we construct a network with 8 peer nodes and 1 contract node. All peers communicate with the one contract node since the blockchain network deploys an equal *wrapper* to all nodes. Since the REE evaluates in the same environment as MECAT, the *wrappers* communicate with the same embedded device. We measured the transaction latency and throughput by increasing clients from 4 to 20. During 3 minutes, each client invokes 10 transactions per second.

The top row of Figure 8 shows the write transaction overheads. Although the latency is 2.45x increased, the throughput of MECAT follows the Native before the 16 clients. When the client number reaches 16, the throughput is dropped by 9.08%, but the latency is increased by 2.91x since the latency of Native also increased by 41.54x. In addition, the throughput has grown with the number of clients, and the average power overhead also increased to process the transactions. Compared with REE, the latency has increased by about 2.45x and 2.72x, at the 12 clients and 16 clients. Nevertheless, the average power is slightly increased by 3.86% and 5.00%, since the processing time of transactions is the same regardless of the number of clients.

The bottom row of Figure 8 shows the read transaction overheads which tend to be like write transactions. However, since the query does not update the ledger, it has a shorter execution time than the write transactions, allowing MECAT to perform with 20 clients. In addition, Native shows thresholds with 120 transactions per second across more than 12 clients, a similar trend in other environments.

In the evaluation environment, MECAT processes requested transactions sequentially in the contract node. Therefore, we expect that if transaction requests can be processed simultaneously the performance of MECAT can reach the mostly same as the Native.

## VIII. DISCUSSION

The current prototype of MECAT is based on the Fabric network, supporting smart contracts. However, as MECAT follows a modular design principle, it can be easily extended to various blockchain networks without re-implementing all components (i.e., *wrapper*, *proxy*, and *memory-safe chaincode*). Specifically, if a blockchain network provides smart contracts based on a different programming language, extending MECAT can be achieved as follows. First, the smart contract has to be modified to call MECAT APIs for the *wrapper*. Since MECAT provides *MECAT_Message* supporting communication between the *wrapper* and *proxy* such modifications would be straightforward. Then, the actual contract operations for each subcommand of the smart contract must move to the secure world in TrustZone and be written in Rust. Even in this case, developers can effortlessly port their code, as MECAT provides APIs for context management. It is noteworthy that the MECAT *proxy* remains unchanged, allowing developers to extend to different blockchain networks without rewriting the message transmission process between the *wrapper* and *memory-safe chaincode*.

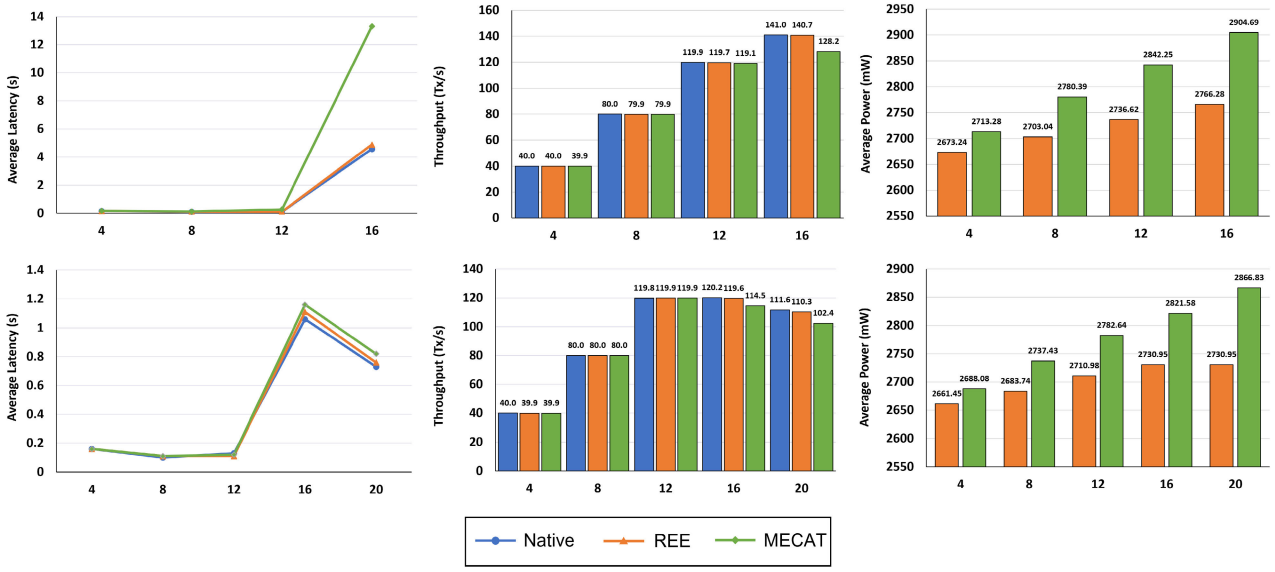


**FIGURE 8.** Network evaluation result of MECAT with different numbers of clients. The top row shows the result of write transactions and the bottom row shows the result of read transactions.

In addition, MECAT can be guaranteeing the memory safety of the smart contracts, nevertheless, the vulnerabilities of smart contracts inherent in the business logic can exist. These can be mitigated by vulnerability detection methods such as fuzzing which is our orthogonal approach.

## IX. CONCLUSION

In this study, we introduced MECAT, which supports memory-safe confidential smart contracts. In MECAT, smart contracts are written in Rust, a memory-safe language, and executed in TEE. By doing this, MECAT is free from memory vulnerabilities in the smart contract. As a result, MECAT provides compatibility with its modular design and provides an API to make it easy to use. In the evaluation, we show the energy efficiency and practical performance of MECAT in both single-node environment and network made with 8 peers. As our result shows MECAT can process 16 clients in the Hyperledger Fabric network and only incurs 5.00% of power overhead. However, in the current environment deploying one single contract node in the network. It is expected that the network performance of MECAT can be improved through multi-node deployment or multi-threading support, we leave this as future work.

## REFERENCES

[1] B. M. Yakubu, R. Latif, A. Yakubu, M. I. Khan, and A. I. Magashi, "RiceChain: Secure and traceable Rice supply chain framework using blockchain technology," *PeerJ Comput. Sci.*, vol. 8, p. e801, Jan. 2022.

[2] B. M. Yakubu, M. I. Khan, N. Javaid, and A. Khan, "Blockchain-based secure multi-resource trading model for smart marketplace," *Computing*, vol. 103, no. 3, pp. 379–400, Mar. 2021.

[3] M. Waleed, R. Latif, B. M. Yakubu, M. I. Khan, and S. Latif, "T-smart: Trust model for blockchain based smart marketplace," *J. Theor. Appl. Electron. Commerce Res.*, vol. 16, no. 6, pp. 2405–2423, Sep. 2021.

[4] R. Li, Q. Wang, Q. Wang, D. Galindo, and M. Ryan, "SoK: TEE-assisted confidential smart contract," *Proc. Privacy Enhancing Technol.*, vol. 2022, no. 3, pp. 711–731, Jul. 2022.

[5] H. F. Atlam and G. B. Wills, *Intersections Between IoT and Distributed Ledger* (Advances in Computers). vol. 115. Amsterdam, The Netherlands: Elsevier, 2019, pp. 73–113.

[6] B. M. Yakubu, M. I. Khan, A. Khan, F. Jabeen, and G. Jeon, "Blockchain-based DDoS attack mitigation protocol for device-to-device interaction in smart home," *Digit. Commun. Netw.*, vol. 9, no. 2, pp. 383–392, Apr. 2023.

[7] B. M. Yakubu, M. I. Khan, A. Khan, A. Anjum, M. H. Syed, and S. Rehman, "A privacy-enabled, blockchain-based smart marketplace," *Appl. Sci.*, vol. 13, no. 5, p. 2914, Feb. 2023.

[8] T. Li, Y. Fang, Z. Jian, X. Xie, Y. Lu, and G. Wang, "ATOM: Architectural support and optimization mechanism for smart contract fast update and execution in blockchain-based IoT," *IEEE Internet Things J.*, vol. 9, no. 11, pp. 7959–7971, Jun. 2022.

[9] 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.

[10] S. Pinto and N. Santos, "Demystifying arm TrustZone: A comprehensive survey," *ACM Comput. Surv.*, vol. 51, no. 6, pp. 1–36, Nov. 2019.

[11] Z. Jian, Y. Lu, Y. Qiao, Y. Fang, X. Xie, D. Yang, Z. Zhou, and T. Li, "TSC-VEE: A TrustZone-based smart contract virtual execution environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 34, no. 6, pp. 1773–1788, Jun. 2023.

[12] C. Müller, M. Brandenburger, C. Cachin, P. Felber, C. Göttel, and V. Schiavoni, "TZ4Fabric: Executing smart contracts with ARM TrustZone: (Practical experience Report)," in *Proc. Int. Symp. Reliable Distrib. Syst. (SRDS)*, Sep. 23, 2020, pp. 31–40. [Online]. Available: https://ieeexplore.ieee.org/document/9251926

[13] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. D. Caro, D. Enyeart, C. Ferris, G. Laventman, and Y. Manevich, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proc. 13th EuroSys Conf.*, 2018, pp. 1–15.

[14] D. Cerdeira, J. Martins, N. Santos, and S. Pinto, "ReZone: Disarming TrustZone with TEE privilege reduction," in *Proc. 31st USENIX Secur. Symp. (USENIX Secur.)*, 2022, pp. 2261–2279.

[15] N. Matsakis and I. K. Felix, "The rust language," in *Proc. ACM SIGAda Annu. Conf. High Integrity Lang. Technol.*, Oct. 2014, pp. 103–104. [Online]. Available: http://dl.acm.org/citation.cfm?id=2663188

[16] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone explained: Architectural features and use cases," in *Proc. IEEE 2nd Int. Conf. Collaboration Internet Comput. (CIC)*, Nov. 2016, pp. 445–451.

[17] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou, "Hawk: The blockchain model of cryptography and privacy-preserving smart contracts," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2016, pp. 839–858.
[18] V. Costan and S. Devadas, "Intel sgx explained," Cryptol. ePrint Archive, Tech. Paper 2016/086, 2016. [Online]. Available: https://eprint.iacr.org/2016/086
[19] R. Cheng, F. Zhang, J. Kos, W. He, N. Hynes, N. Johnson, A. Juels, A. Miller, and D. Song, "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *Proc. IEEE Eur. Symp. Secur. Privacy (EuroS&P)*, Jun. 2019, pp. 185–200. [Online]. Available: https://ieeexplore.ieee.org/document/8806762
[20] R. Yuan, Y.-B. Xia, H.-B. Chen, B.-Y. Zang, and J. Xie, "ShadowEth: Private smart contract on public blockchain," *J. Comput. Sci. Technol.*, vol. 33, no. 3, pp. 542–556, May 2018.
[21] M. Russinovich, E. Ashton, C. Avanessians, M. Castro, A. Chamayou, S. Clebsch, M. Costa, C. Fournet, M. Kerner, and S. Krishna, "CCF: A framework for building confidential verifiable replicated services," Microsoft, Redmond, WA, USA, Tech. Rep. MSR-TR-201916, 2019.
[22] Y. Yan, C. Wei, X. Guo, X. Lu, X. Zheng, Q. Liu, C. Zhou, X. Song, B. Zhao, and H. Zhang, "Confidentiality support over financial grade consortium blockchain," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2020, pp. 2227–2240.
[23] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti, "Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric," 2018, *arXiv:1805.08541*.
[24] H. Yin, S. Zhou, and J. Jiang, "Phala network: A confidential smart contract network based on polkadot," Phala Network, Singapore, 2019.
[25] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "SoK: Understanding the prevailing security vulnerabilities in TrustZone-assisted TEE systems," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2020, pp. 1416–1432. [Online]. Available: https://ieeexplore.ieee.org/document/9152801
[26] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "vTZ: Virtualizing ARMTrustZone," in *Proc. 26th USENIX Secur. Symp. (USENIX Secur.)*, 2017, pp. 541–556.
[27] L. Guan, P. Liu, X. Xing, X. Ge, S. Zhang, M. Yu, and T. Jaeger, "TrustShadow: Secure execution of unmodified applications with ARM TrustZone," in *Proc. 15th Annu. Int. Conf. Mobile Syst., Appl., Services*, Jun. 2017, pp. 488–501.
[28] D. Kwon, J. Seo, Y. Cho, B. Lee, and Y. Paek, "PrOS: Light-weight privatized se cure OSes in ARM TrustZone," *IEEE Trans. Mobile Comput.*, vol. 19, no. 6, pp. 1434–1447, Jun. 2020.
[29] F. Brasser, D. Gens, P. Jauernig, A.-R. Sadeghi, and E. Stapf, "SANCTUARY: ARMing TrustZone with User-space enclaves," in *Proc. NDSS*, 2019, pp. 1–15.
[30] S. Wan, M. Sun, K. Sun, N. Zhang, and X. He, "RusTEE: Developing memory-safe ARM TrustZone applications," in *Proc. Annu. Comput. Secur. Appl. Conf.*, Dec. 2020, pp. 442–453.
[31] K. Ryan, "Hardware-backed heist: Extracting ECDSA keys from Qualcomm's TrustZone," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 181–194.
[32] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "VoltJockey: Breaching TrustZone by software-controlled voltage manipulation over multi-core frequencies," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2019, pp. 195–209.
[33] W. Liu, "Research on DoS attack and detection programming," in *Proc. 3rd Int. Symp. Intell. Inf. Technol. Appl.*, vol. 1, Nov. 2009, pp. 207–210.
[34] Apache. *Incubator-Teaclave-TrustZone-SDK*. Accessed: Nov. 17, 2023. [Online]. Available: https://github.com/apache/incubator-teaclave-trustzone-sdk
[35] Hyperledger. *Hyperledger-Caliper*. Accessed: Nov. 17, 2023. [Online]. Available: https://github.com/hyperledger/caliper
[36] J. Amacher and V. Schiavoni, "On the performance of ARM TrustZone: (Practical experience report)," in *Proc. Distrib. Appl. Interoperable Syst.: 19th IFIP WG 6.1 Int. Conf., DAIS, Held 14th Int. Federated Conf. Distrib. Comput. Techn. (DisCoTec)* Kongens Lyngby, Denmark: Springer, 2019, pp. 133–151.

**SEONGHWAN PARK** received the B.S. degree in computer engineering from Dongseo University, South Korea, in 2021. He is currently pursuing the Ph.D. degree with Pusan National University, Busan, Republic of Korea. His research interests include system security and H/W architecture.

**HAYOUNG KANG** received the B.S. degree in mathematics from Pusan National University, South Korea, in 2024, where she is currently pursuing the master's degree in computer engineering. Her research interest includes system security.

**SANGHUN HAN** received the B.S. degree in computer engineering from Pukyong National University, South Korea, in 2020. He is currently pursuing the master's degree in computer engineering with Pusan National University, Republic of Korea. His research interests include compilers, software optimization, and cyber security.

**JONGHEE M. YOUN** received the B.S. degree from the School of Electronics and Electrical Engineering, Kyungpook National University, Daegu, South Korea, in 2003, and the Ph.D. degree in electrical engineering and computer science from Seoul National University, in 2011. He is currently an Associate Professor with the Department of Computer Engineering, Yeungnam University. His research interests include compiler, software optimization, embedded systems, mobile computing, and cyber security.

**DONGHYUN KWON** received the B.S. and Ph.D. degrees in electrical and computer engineering from Seoul National University, South Korea, in 2012 and 2019, respectively. He is currently a Professor with the School of Computer Science and Engineering, Pusan National University, South Korea. His research interest includes system security against various types of threats.