TSC-VEE: A TrustZone-Based Smart Contract Virtual Execution Environment

Zhaolong Jian, Ye Lu, Youyang Qiao, Yaozheng Fang, Xueshuo Xie, Dayi Yang, Zhiyuan Zhou, and Tao Li, *Member, IEEE*

***Abstract*—TrustZone as a trusted execution environment (TEE) has been proven to preserve the confidentiality of blockchain transactions supported by smart contracts. Despite some academic effort, TrustZone can only support limited languages for now. The lack of the corresponding execution environment for smart contracts seriously hinders blockchain applications from directly running on TrustZone. In this paper, we design the first virtual execution environment named TSC-VEE for performing Solidity smart contracts on TrustZone, to the best of our knowledge. TSC-VEE can be decomposed into fourfold: (1) an instruction set adapted to the isolation and world switching mechanism of TrustZone. (2) a runtime memory management mechanism that provides a pair of instructions with the corresponding processing mechanism to allocate and release the work memory. (3) a hybrid granularity resource analysis algorithm which computes and records the value of maximum stack height and static gas cost through bytecode pre-execution, avoiding runtime overflow and invalid computations. (4) a cross-isolation-environment prefetching approach that supports loading and storing the storage data from the normal world into the secure world on TrustZone before execution, thus avoiding switching the world state frequently at runtime. Extensive experimental results show that TSC-VEE can perform smart contracts correctly and efficiently on TrustZone. Compared with the most commonly used Ethereum client—*Geth*, TSC-VEE achieves execution performance improvements by $9.29\times$. We also implement the Ethereum virtual machine—*evmone* on TrustZone. TSC-VEE can reduce the latency by 12.63% with our optimization techniques, and decrease the work memory footprint by 22.95% on average when executing various scale contracts.**

***Index Terms*—Blockchain, smart contract, solidity program language, TrustZone, virtual execution environment.**

Manuscript received 19 September 2022; revised 24 March 2023; accepted 29 March 2023. Date of publication 6 April 2023; date of current version 8 May 2023. This work was supported in part by the CCF-AFSG Research Fund under Grant CCF-AFSG RF20210031, in part by the CCF-Huawei Populus Grove Fund under Grant CCF-HuaweiTC2022005, in part by the National Natural Science Foundation under Grant 62002175, in part by the Open Project Fund of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences under Grant CARCHB202016, and in part by the Open Project Foundation of Information Security Evaluation Center of Civil Aviation, Civil Aviation University of China under Grant ISECCA-202102. Recommended for acceptance by Y. Yang. *(Corresponding author: Ye Lu.)*

Zhaolong Jian is with the College of Computer Science, Nankai University, Tianjin 300350, China, also with the Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province, Hangzhou 310058, China, and also with the Tianjin Key Laboratory of Network and Data Science Technology, Tianjin 300350, China (e-mail: jianzhaolong@mail.nankai.edu.cn).

Ye Lu is with the College of Computer Science, Nankai University, Tianjin 300350, China, also with the College of Cyber Science, Nankai University, Tianjin 300350, China, also with the Institute of Systems and Networks, Nankai University, Tianjin 300350, China, also with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100045, China, and also with the Tianjin Key Laboratory of Network and Data Science Technology, Tianjin 300350, China (e-mail: luye@nankai.edu.cn).

Youyang Qiao and Yaozheng Fang are with the College of Computer Science, Nankai University, Tianjin 300350, China, and also with the Tianjin Key Laboratory of Network and Data Science Technology, Tianjin 300350, China (e-mail: youyangqiao@mail.nankai.edu.cn; fyz@mail.nankai.edu.cn).

Xueshuo Xie is with the College of Computer Science, Nankai University, Tianjin 300350, China, also with the College of Cyber Science, Nankai University, Tianjin 300350, China, also with the Key Laboratory of Blockchain and Cyberspace Governance of Zhejiang Province, Hangzhou 310058, China, and also with the Tianjin Key Laboratory of Network and Data Science Technology, Tianjin 300350, China (e-mail: xueshuoxie@nankai.edu.cn).

Dayi Yang and Zhiyuan Zhou are with the Blockchain Platform Division, Ant Group, Beijing 100000, China (e-mail: dayi.ydy@antgroup.com; wenzhang.zzy@antgroup.com).

Tao Li is with the College of Computer Science, Nankai University, Tianjin 300350, China, also with the College of Cyber Science, Nankai University, Tianjin 300350, China, also with the State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100045, China, and also with the Tianjin Key Laboratory of Network and Data Science Technology, Tianjin 300350, China (e-mail: litao@nankai.edu.cn).

Digital Object Identifier 10.1109/TPDS.2023.3263882

I. Introduction

THE wide application of smart contracts has greatly boosted the development of blockchain [1], [2]. Smart contracts are essentially executable codes stored on the blockchain [2], [3]. Their characteristics such as transparency and immutability are a double-edged sword. Smart contracts with these characteristics enable blockchain to execute digital agreements between untrusted entities. They have been widely used as the application carrier and serve many fields, such as the Internet of Things, access control, and certificate audit [4], [5], [6]. However, smart contracts stored publicly on blockchain ledgers expose the assets they carry to serious risks. Due to confidentiality not being guaranteed, the attacker can obtain the execution logic of smart contracts through bytecode decompilation to create targeted attacks. Such attacks have caused millions of dollars in economic losses [7].

Recently, building confidential smart contracts with the assistance of trusted execution environments (TEEs) has been a general solution [8]. TEEs can provide a special environment for trusted code execution through the isolation mechanism of software and hardware, such as Intel SGX [9], ARM TrustZone, AMD SME/SEV, RISC-V Keystone, and Penglai Enclave [10], [11], [12], [13]. Though promising, migrating the mainstream smart contracts to TEEs is a complex and difficult task, mainly due to the incompatibility between the languages and environments that the smart contracts usually rely on and those provided by TEEs. The difficulty of such migration is especially pronounced on TrustZone.

TrustZone is the widely supported TEE on ARM-based embedded devices and cloud servers. Recently, embedded devices have become an important carrier of smart contracts [14], [15]. Among them, more than 90% of the embedded devices are equipped with ARM chips [16]. The ubiquity of TrustZone makes it an attractive TEE base for preserving the confidentiality of smart contracts [17], [18]. However, TrustZone commonly uses OP-TEE[1] as the security operating system, which limits the secure resources to minimize the trusted computing base (TCB) and reduce the attack surface to improve security. OP-TEE only provides a C language runtime environment [19]. It implies that TrustZone can only support trusted applications (TAs) written in the limited language until now. The existing mainstream smart contracts are programmed in domain-specific languages such as Solidity [20]. There are more than 30 million smart contracts on Ethereum. Most of these mainstream contracts are programmed in Solidity [21], with an average of several hundred lines of code per contract [22], [23]. These smart contracts cannot be directly migrated and executed on TrustZone on account of no corresponding execution environment support. Although rewriting these smart contracts can help to convert them into C language, the migration with about several billion lines of code requires tens of thousands of man-year labor works, which is obviously not a reasonable choice. Minimal support for the virtual execution environment is necessary, although it brings a limited increase in TCB [17].

In this paper, we design a new virtual execution environment named TSC-VEE, for performing mainstream smart contracts programmed in Solidity on TrustZone. TSC-VEE can be treated as the computation core of confidential smart contracts. TSC-VEE aims to provide a highly efficient execution environment and optimization mechanisms for smart contract execution. To do this, we should overcome several difficulties as follows. First, the existing Solidity instruction set for smart contracts does not match the program execution mode of TrustZone and OP-TEE. The native instruction will cause world switches and consume lots of time. The blockchain client is running on the rich execution environment (REE) side (the normal world), while the instruction execution is on the TEE side (the secure world). Therefore, the common instruction for persistent data access needs to penetrate the contract execution environment, switch the world state, and be completed via the host application. This time-consuming process makes data access across the world very expensive. Second, the runtime data will be enlarged along with smart contract execution, but TrustZone has very limited memory resources. The smart contract programmed by Solidity language has very rudimentary memory management currently, lacking the memory recycling mechanism to release the work memory at runtime [24]. The size of the secure memory available for contract execution on TrustZone is usually limited to the ten MB level [25]. When the secure memory cannot meet the memory footprint of the execution, the smart contract will not be executed solidly. Third, frequent resource detection during contract execution slows down execution performance. The instructions in the contract bytecode will be performed one by one during execution. Before executing each instruction, it is necessary to detect whether there is a stack overflow or out-of-gas exception to ensure the correctness of contract execution. With this detection method, the number of runtime detection can be increased largely, and the detection latency can account for 20% or even more of the instruction execution latency. Also, this method involves meaningless instruction execution when an exception occurs.

To the best of our knowledge, TSC-VEE is the first virtual execution environment on TrustZone that can support performing the Solidity smart contracts. In this paper, our concrete contributions can be summarized as follows:

- We design a specific instruction set for describing the atomic operations of smart contracts according to the mechanism of TrustZone and OP-TEE. The low-level interpretation of the instruction set can support mainstream contracts performing on TrustZone.
- We present a runtime memory management (RMM) mechanism with a pair of special instructions to allocate the work memory from the operating system level, and dynamically release them at runtime. RMM facilitates TSC-VEE to reduce the memory footprint by 22.95% on average, with only 0.97% additional latency overhead when performing contracts of large size.
- We design a hybrid granularity bytecode analysis (HGBA) algorithm in TSC-VEE. HGBA can reduce the number of runtime detection through bytecode pre-execution without compromising execution correctness. So TSC-VEE can decrease the execution latency by about 6.04% on average with the number of execution increases.
- We propose a cross-isolation-environment prefetching (CIEP) method, which can load and store the persistent storage data of blockchain into the secure world in advance to avoid world switching at runtime. CIEP can help TSC-VEE reduce smart contract execution latency by 7.48% on average.
- We implement TSC-VEE on the Raspberry Pi 3B+ and verify the complete smart contract execution process on TrustZone. Compared with the Ethereum client—*Geth*, TSC-VEE achieves $9.29\times$ execution performance improvement. Compared with *evmone* we implemented on TrustZone as baseline, the execution latency of TSC-VEE can also be reduced by 12.63%, which is roughly the same performance in the normal environment.

II. Background and Motivation

In this section, we draw our motivation and detailed challenges in designing TSC-VEE from three aspects. We first introduce the smart contract, point out the requirements for trusted execution, and make a comparison with related works. Then, we discuss the program execution mechanism on TrustZone and analyze the shortcomings caused by the lack of the execution environment. At last, we give the main challenges of designing TSC-VEE by breaking down analysis of smart contract execution on TrustZone in detail.

[1] https://optee.readthedocs.io/en/latest/

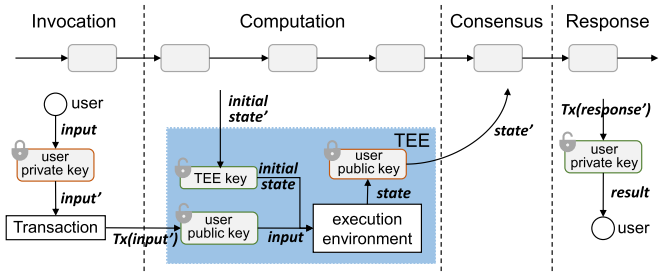


Fig. 1. Confidential smart contract workflow.

A. Solidity Smart Contract

The concept of smart contracts was first described by Nick Szabo: A smart contract is a computerized transaction protocol that executes the terms of a contract [26]. The development of blockchain technology makes the smart contract possible. The blockchain-based smart contracts usually take the form of generic programs. They can be automatically executed and stored in a blockchain network [2], [3]. Developers can write and deploy any Turing-complete program on the blockchain network. In a decentralized smart contract system, the consensus system enforces the autonomous execution of the contract. No single entity or small group of entities can interfere with the contract execution. Therefore, smart contracts can be executed once the trigger conditions are satisfied and cannot be modified after deployement [15], [27]. The self-executing mode of smart contracts eliminates the need for trusted intermediaries or reputation systems to reduce transaction risk. Since smart contracts are integrated into the blockchain, they inherit the blockchain features of availability, transparency, immutability, and integrity. These advantages make smart contracts widely used in the asset transactions and development of various DAPPs.

The most famous smart contract implementation is Ethereum [28]. In 2020 alone, developers on the Ethereum mainnet have created a total of 10.7 million smart contracts involving tens of billions of dollars in assets [7]. Ethereum provides a runtime environment called the Ethereum Virtual Machine (EVM). The smart contracts are compiled into EVM bytecode and deployed to the Ethereum network via transactions. These smart contracts are written in Solidity language. Solidity is a contract-oriented high-level programming language created to implement mainstream smart contracts. When these contracts run on EVM, Ethereum will utilize Gas, which is used for solving the halting problem, to limit their computational works and operations.

B. Confidential Smart Contract Execution

Some research has focused on using TEE to provide the capability of confidential smart contracts. The typical confidential smart contract workflow is divided into four stages, as shown in Fig. 1. First, the user initiates the confidential smart contract invocation by submitting a transaction, which carries the input data encrypted by the user's private key. Second, after receiving the invocation request, TEE decrypts data of the transaction using the user's public key. Then, the TEE loads the encrypted source code and state from the blockchain network and decrypts them using the TEE service key. Afterward, TEE executes the smart contract in the execution environment, outputs the encrypted result, and sends the ciphertext of the new state to the blockchain network. Third, the nodes of the blockchain network run the consensus algorithm to confirm the execution result of the transaction and the new state. Fourth, the user obtains the ciphertext of the state from the transaction response and can obtain the final plaintext by decrypting data using the user's private key.

Following the above execution model, Confidential Consortium Framework (CCF) [29], CONFIDE [30] and FPC [31] equip each blockchain node with TEE. The smart contract executes in the isolated area of TEE, and the encrypted result is broadcast to peer nodes for consensus. These solutions are integrated into their consortium blockchain, respectively. Hawk [32], ShadowEth [33], Ekiden [34], Fastkitten [35], TZ4Fabric [17] and Phala [36] decouple the smart contract execution from the blockchain system and execute the contracts off-chain as a separate layer. The user pushes the smart contract to the off-chain TEE platform for execution, and only uploads the result to the blockchain for consensus.

Compared to this research TSC-VEE focuses on the computation stage of the confidential smart contract rather than the entire process. TSC-VEE is dedicated to filling the gap in smart contract execution environment on TrustZone to benefit from the ubiquity of TrustZone, and improving the performance of the execution environment. In terms of smart contract execution, most of the previous works use SGX as the underlying TEE. The techniques of these works cannot be directly applied to perform smart contracts of Solidity language on TrustZone. For example, ShadowETH [33] and TZ4Fabric [17] are oriented to contracts written or rewritten in C/C++. They are unable to execute Solidity language bytecode. CCF [29] and Ekiden [34] provide execution environments for Solidity smart contracts, but these execution environments cannot be applied to TrustZone for two reasons. First, TrustZone only provides a C language runtime environment, but Ekiden and CCF implementation require Rust and C++, respectively. Second, the SGX hardware mechanism by Enclave is different from TrustZone. The size of EPC memory on SGX is usually at the hundred MB level, while the secure memory on TrustZone-based TEEs is usually at the ten MB level. SGX enables the dynamic creation of enclaves within the virtual address space of user-mode processes, while TrustZone provides two world states with system-wide hardware-enforced isolation. Therefore, the Ekiden and CCF approach cannot meet the challenge of the time-consuming world switching. In addition, to the best of our knowledge, Ekiden and CCF do not have any special performance optimization mechanisms [37]. In terms of security, TSC-VEE follows the process of the computation stage shown in Fig. 1, encrypting the input and output data during execution. TSC-VEE can keep the same security property as the above research.

Moreover, the state-of-the-art TEE (e.g., SGX v2 and ARM CCA) provides support for large secure memory, making it easier to execute the smart contract on TEE. However, SGX v2 increases the size of secure memory by adding EPC memory, which increases the cost and is only applicable to specific devices [38]. The device supporting ARM CCA is a long time away

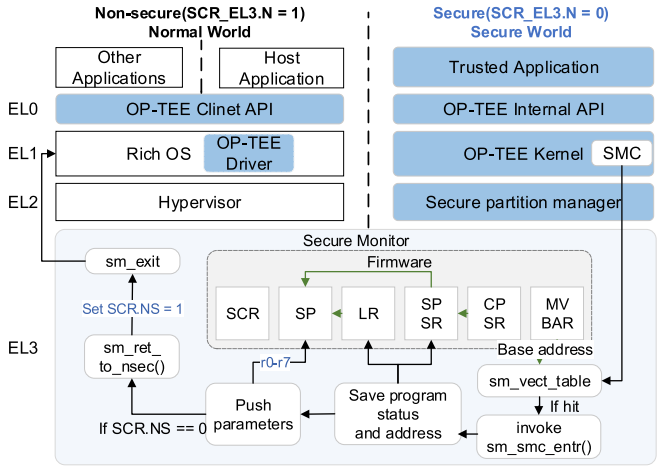


Fig. 2. Architecture and the typical execution process of trusted application on TrustZone.

from commercial use [39]. Although these TEEs can provide more secure memory, the efficiency of memory usage still needs to be improved.

Through these comparisons, we can see that the confidential smart contract execution on the cloud servers with Intel SGX has been studied extensively, while the embedded devices still lack execution environment support[8]. The embedded devices always serve as the data source and need TEEs to provide the confidentiality guarantee for the smart contracts. These facts motivate us to design a new execution environment for performing mainstream Solidity smart contracts on embedded devices with TrustZone.

C. Program Execution Mechanism on TrustZone

TrustZone Architecture. TrustZone is the security architecture in the ARM processor. As shown in Fig. 2, TrustZone provides two execution environments with system-wide hardware-enforced isolation between them. The TEE part is called the secure world, and the other part is called the normal world. The processor can be in one of two states: secure and non-secure. World switching between the two states happens via a secure monitor call (SMC). The system resources on TrustZone are strictly isolated: the normal world cannot access the resources reserved for the secure world, such as memory, peripherals, etc. OP-TEE is a popular open-source secure OS designed as the companion to a non-secure Linux kernel running on ARM [25]. It is designed primarily to rely on the ARM TrustZone technology as the underlying hardware isolation mechanism. OP-TEE follows the TEE architecture and provides API for both REE and TEE sides that are standardized by the GlobalPlatform.

World Switch. As shown in Fig. 2, software based on the ARM architecture is divided into four exception levels: EL0-EL3, and their corresponding privilege levels are increasing. The application runs at EL0, and the operating system runs at EL1. EL2 is used by the hypervisor or secure partition manager. EL3 is reserved by low-level firmware and security code. After the SMC is sent from the OP-TEE kernel, the processor will trigger an exception to enter the monitor mode at the highest privilege level, which can process the world switching. The SMC monitor first obtains the base address of the abnormal interrupt vector table from the MVBAR register and finds the abnormal handling function of the SMC. The state of the program saved in the CPSR register will be stored in the SPSR register, and the LR register saves the return address of the subroutine. The values in r0-r7 registers will be pushed onto the stack. Then the monitor will judge the current world state by the value of the *SCR.NS* flag in the SCR register. When the value is 0, the current state is the secure world. The monitor will save the context of the secure world and then obtain the context of the normal world. Finally, the monitor switches to the secure world by invoking the function *sm_ret_to_nsec()* to set the value of the *SCR.NS* flag to 1.

Execution Mechanism. Due to the lack of execution environment, the only way to execute Solidity smart contracts is to implement the smart contract as a trusted application on TrustZone. Under the OP-TEE secure operating system, the host application running in the normal environment and the TA in the secure world complete the application call process together. The host application and TA are programmed in C language and comply with the OP-TEE client API and OP-TEE internal API, respectively. These APIs enable communication between the host application and the trusted application. This kind of communication is supported through world switching by SMC. OP-TEE also provides a memory area on the REE side as the shared memory. It can be accessed by both the REE side and the TEE side to assist the data interaction between them.

This execution mode has obvious disadvantages. First, the developers need to reconstruct each smart contract from Solidity language to a TA in C language. The developers should also provide a host application to support completing the TA invoke and data interaction in the normal world. This execution mode greatly increases the cost of smart contract development. Second, the target machine will use the original build key to sign the TA during compilation for security reasons. Therefore, the contracts must be compiled into TAs and installed on the target machine before the contract execution. However, smart contract applications usually need to be updated frequently to adapt to changes in application logic or improve security [15]. This execution mode will bring a high additional latency overhead.

D. Analysis on Challenges

In order to solve the problems caused by the lack of execution environment on TrustZone, we propose TSC-VEE aiming at supporting mainstream smart contracts migrating and performing on TrustZone. TSC-VEE is an independent execution environment module like evmone[2], aleth-interpreter[3], etc. TSC-VEE operates on a different level than the consensus and network mechanisms the consensus and network mechanism and will not affect the decentralization characteristics or execution workflow of the underlying blockchain system. There have been some works devoted to embedding execution environments for different languages into TrustZone, such as TLR [40] and RusTEE [19]. By embedding these runtime environments, TrustZone can obtain the ability to execute programs in different languages and provide trusted code execution. These works further inspire and

[2]https://github.com/ethereum/evmone
[3]https://github.com/ethereum/aleth/tree/master/libaleth-interpreter

motivate us to introduce Solidity smart contract execution into TrustZone. Compared with TLR and RusTEE, TSC-VEE not only adapts the execution environment to the TrustZone but also aims to make more effort to optimize the execution environment itself to improve performance. There are three main challenges we need to overcome as follows:

First, the existing instruction set of Solidity and the corresponding interpreter cannot match the execution mechanism of TrustZone. The data access instructions will cause world switches, resulting in high latency. As aforementioned, the smart contract is executed on the TEE side, but the persistent storage data required during execution is located at the blockchain stateDB on the REE side. Therefore, cross-world data accesses like SSTORE or SLOAD instruction need to penetrate the execution environment. Specifically, for data loading operation, the corresponding data in the blockchain stateDB on REE side should be first copied to the shared memory via the host application. Until the state has been switched back to the secure world, TEE can access the data in the shared memory. According to our testing, such cross-world data access operation can take about 6,000 clock cycles.

Second, TrustZone-based TEEs have very limited memory resources. On the embedded development board, the available secure memory may be only ten MB level, while the typical memory footprint of executing smart contracts will increase from hundreds of KB to hundreds of MB according to the computational complexity. During execution, the bytecode, function parameters, and intermediate data generated at runtime will occupy secure memory space. Since there is no memory recycling mechanism in Solidity language, the size of intermediate data in the work memory will keep growing at runtime. Especially when the amount of function parameters or the calculation scale is large, insufficient memory will cause the contract execution to fail.

Third, frequent resource detection during contract execution will increase the execution latency. During contract execution, the instructions in the bytecode are executed serially. When executing an instruction, the execution environment first computes the stack height and gas cost and then executes the instruction function. Such detection can ensure the correctness of contract execution and help developers locate the position of instruction where the exception occurs. However, this detection method does not consider the difference between stack and gas at the point of the execution importance and the corresponding exception frequency, thereby increasing the detection times. When stack overflow or out-of-gas exception occurs, contract execution will be terminated, and the results are also invalid.

III. Basic Design of TSC-VEE

In this section, we propose the basic design of TSC-VEE, to the best of our knowledge, the first runtime environment for performing smart contract bytecode in Solidity language on TrustZone and speeding up the execution efficiency. Here, we first introduce the threat model and then present the overall architecture of TSC-VEE. We then give the dedicated instruction set design and explain the dynamic memory management mechanism, RMM. Next, we introduce the hybrid granularity bytecode analysis algorithm. In the end, we propose the cross-isolation-environment prefetching to further improve smart contract execution efficiency.

A. Threat Model

TSC-VEE is designed for devices equipped with Arm TrustZone and oriented to the common scenario recognized in related work [17], [19]. We assume all software components of TrustZone (including bootloader, firmware, security monitor, and OP-TEE OS) are trusted, while the OS and user space in the normal world are not trusted. We also assume the users have secure methods to collect and process the input data. We encrypt the input and output data and rely on the hardware isolation mechanism of TrustZone to provide the confidentiality guarantee for TSC-VEE (implemented as a TA). TSC-VEE thus can provide security protection of smart contract execution from the input data's arrival to the output results' leave. Attacks unveiled against TrustZone itself like side-channel attacks [41], [42] are out of the scope of our work. These attacks can be mitigated by some orthogonal solutions [43]. In addition, TSC-VEE cannot protect against rollback attacks, since it is deployed on Ethereum with non-final decision consensus [31].

B. Architecture Overview

As shown in Fig. 3, Our TSC-VEE is designed to be a stack-based program virtual machine and as a TA on the TEE side. There are three key parts of executing smart contracts in TSC-VEE: Data Area, Instruction Set, and Interpreter. Data Area includes the stack and the work memory. The stack is an operand stack and is responsible for storing the operands needed during execution. Its word length is 256 bits. The work memory is responsible for storing various types of data and the return values at runtime. Before being used as operands, the runtime data in the work memory and the state data in the blockchain storage will first be loaded into the stack. The instruction set contains all the instructions supported by TSC-VEE. TSC-VEE provides a jump table that records the correspondences from opcodes to operations or from operations to instruction functions.

The instruction interpreter is the core of TSC-VEE, responsible for the bytecode interpretation and execution of smart contracts. The interpreter consists of four main execution stages. First, the program counter fetches the opcode from the input bytecode stored at the TEE side. The opcode is a two-digit hexadecimal number. Second, in the decoding stage, according to the jump table of the instruction set, the interpreter converts the opcode into an operation. Third, the interpreter analyzes the stack requirement of the instruction and the static gas cost through bytecode pre-execution. The analysis results are cached in the shared memory for the access of the interpreter and avoid increasing the secure memory footprint. During the analysis stage, the corresponding function of each instruction, the parameters of some special types of instruction, and the jump destination of the bytecode block will be recorded and used as the input data for the next execution stage. These three stages will be repeated until all the bytecodes have been loaded. Fourth,

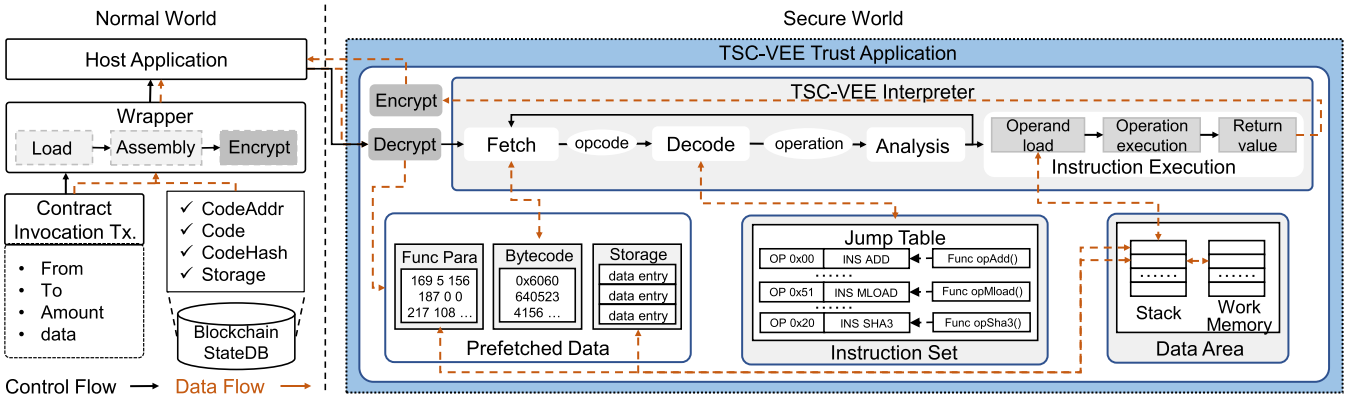


Fig. 3. TSC-VEE architecture overview.

TABLE I
THE KEY INSTRUCTIONS OF TSC-VEE INSTRUCTION SET

Type	Mnemonic & Opcode
Arithmetic	STOP (0x00), ADD (0x01), MOD (0x06)...
Comparison	LT (0x10), EQ (0x04), ZERO (0x05), AND (0x06)...
Crypto	SHA3 (0x20)
Cluster State	CALLVALUE (0x34), CALLDATALOAD (0x35)...
Storage	SLOAD (0x54), SSTORE (0x55), MLOAD (0x51)...
Execution	JUMP (0x56), PC (0x58), JUMPDEST (0x5b)...
Stack	PUSH1 (0x60), DUP1 (0x80), SWAP1 (0x90)...
Logging	LOG0 (0xa0)...
Clusters	RETURN (0xf3)
RMM	MARK (0xB0), FREE (0xB1)

the interpreter executes the instructions sequentially according to the records, dynamically corrects the gas cost, and then returns the execution result.

In addition, the smart contract execution process in TSC-VEE also requires the assistance of the wrapper and host application on the REE side. The wrapper is responsible for data interaction with the blockchain client and provides the required data to the host application. The host application is responsible for processing all data interactions with TA, and serves as the entry point for calling TA.

When executing smart contracts, the application in the normal world will initiate a contract invocation transaction, which contains the contract address, sender, input data, etc. The wrapper will load the bytecode and blockchain state from the blockchain stateDB according to the contract address. These parameters will be packaged and encrypted before sending to the host application. Next, the host application invokes the TA to execute the smart contract. Based on our security assumptions, the workflow of TSC-VEE can ensure the trustworthiness of execution process. Therefore, we focus on optimizing the performance of TSC-VEE in the following sections.

C. Dedicated Instruction Set

Table I shows the instructions of TSC-VEE. The instruction set of TSC-VEE includes 109 general instructions and two special instructions. Each instruction has a mnemonic and corresponding two-digit hexadecimal bytecode. The instruction functions and types in the instruction set are based on the Solidity native instruction set, modifying the underlying implementation mechanism to fit the TrustZone mechanism Instructions within the set are based on the native instruction set of Solidity. The low-level interpretations of these instructions have been refactored and are adapted to the TrustZone and OP-TEE mechanism to support the mainstream contracts.

According to the operation type, the instructions are divided into ten types: 1)$Arithmetic$ instructions include four arithmetic operations, modulo operation, and exponential operation for 256-bit operands. These instructions pop two or three operands from the top of the stack, perform calculations, and then push the result to the stack. 2)$Comparison$ instructions include data size comparison and boolean operations. They are used for operands comparison and jump target judgment, etc. 3)$Crypto$ instruction SHA3 is responsible for the hash calculation of blockchain. 4)$Clusterstate$ instructions are mainly responsible for obtaining the additional data, including sender, input data, and other parameters when the transaction is initiated. 5)$Storage$ instructions include popping operands from the top of the stack, as well as data movement between the stack and TSC-VEE work memory, or between the stack and TSC-VEE storage. 6)$Execution$ instructions mainly include jump instructions and instruction to obtain the current program counter, memory size, and remaining gas. The jump instructions are responsible for processing the jump and verification of the execution flow. 7)$Stack$ instructions include three types. PUSHx instructions push the X bytes (1-32 bytes) data onto the top of the stack. DUPx instructions copy the Xth (1st to 16th) data of the stack and push it onto the top of the stack. SWAPx instructions swap the top element of the stack with the Xth(1st to 16th) element. 8)$Logging$ instructions are used to store logs of different lengths in the blockchain StateDB. 9)$Cluster$ instructions include different contract call mode instructions and the return instruction. 10)RMM instructions include a pair of instructions to create a memory pointer and release memory to complete the RMM mechanism.

It is worth noting that SLOAD and SSTORE instructions involve cross-world data access. Executing these instructions requires considering performing world switching according to

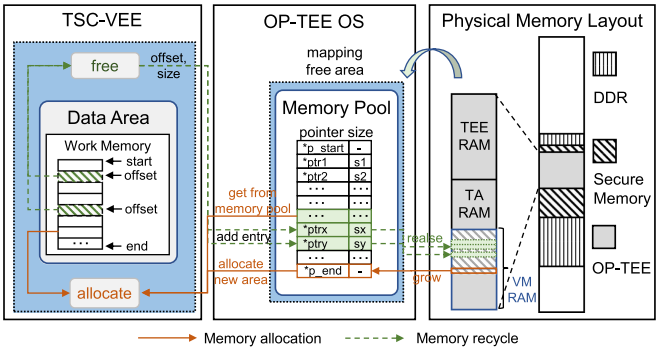


Fig. 4. The processes of memory allocation.

the access mode under CIEP. Moreover, we specially handled the execution mechanism of CALLDATACOPY instruction by setting a position mapping. It can help to avoid repeatedly storing the data which has been preloaded into the TEE side.

D. Runtime Memory Management

Memory management in Solidity is currently very rudimentary and lacks the memory recycling mechanism [24]. Therefore, the work memory footprint will keep growing at runtime and is prone to out-of-memory exceptions. Solidity provides two keywords, $storage$ and $memory$, to specify variable types. Variables of $storage$ type will be persistently stored in the blockchain stateDB. Variables of $memory$ type will be temporarily stored in the work memory and will not be released until the end of execution. Most of the data in the contracts are of $memory$ type to avoid high gas consumption caused by the persistent storage. It is these data that make the work memory(actually the secure memory on TrustZone) footprint sustained growth with contract execution. This feature makes it challenging to execute solidity smart contracts on TrustZone with limited memory. We consider reducing the memory footprint by designing a work memory management mechanism, including memory allocation and memory recycling.

Memory Allocation. For solidity contracts, the choice of accessing work memory is determined during compiling. The compiler will generate the $MLOAD$ and $MSTORE$ instructions at the corresponding location according to the contract logic. When the interpreter encounters instructions for the work memory access or store, it will fetch the value and offset from the stack top. With the memory allocation mechanism, TSC-VEE will send a memory allocation request to the OP-TEE operating system when the work memory space is insufficient. As shown in Fig. 4, we define a fixed-size contiguous physical memory area on TrustZone as a new memory type VM_RAM, which is dedicated to the work memory allocation of TSC-VEE. We register the starting address and size of VM_RAM in OP-TEE and maintain a memory pool pointing to the unused memory area. When OP-TEE receives work memory allocation requests, it will perform the memory allocation algorithm as follows:

1) Expand the work memory area as required when the remaining memory of VM_RAM is sufficient.
2) When the remaining memory cannot meet the requirement, merge the contiguous area in the memory pool and then find an area of the right size.
3) When there is also no available area in the memory pool, move a certain size of data from the starting address of VM_RAM to the shared memory and then release the area.
4) Repeat Step1-3 until the requirement is met.

TABLE II
MEMORY RECYCLING INTERFACE OF TSC-VEE

Interface	Description
Ahead-of-time operations	
mark($parameter$)	Add mark to the memory area of $parameter$
free($parameter$)	Add release mark to $parameters$
Compilation operations	
MARK($parameter$)	Insert $'B0'$ and two-bit $'id'$ into the bytecode block
FREE($parameter$)	Insert $'B1'$ and two-bit $'id'$ into the bytecode block
Runtime operations	
opMark(string id)	Create a mapping to the previous variable in the work memory
opFree(string id)	Release the marked working memory area based on the mapping

Memory Recycling. We consider reducing the runtime memory footprint by releasing the work memory at runtime. The function parameters will be temporarily stored in the work memory as local variables. The data will cause the secure memory footprint to grow with contract execution. We provide the developers with a pair of instructions and the matching processing mechanism to achieve flexible runtime memory management. Table II shows the RMR interface of TSC-VEE. During programming, developers can insert *MARK* and *FREE* marks to the variables according to their actual lifecycle. TSC-VEE also provides a pre-processing approach to help reduce the burden of manual memory management for developers. This approach can identify variable types and analyze their life-cycle according to the pre-defined rules, and insert the marks of function parameters automatically. We recommend that developers fine-tune the marks manually according to the programmatic logic if they want to achieve accurate memory management. At the compilation stage, the marks will be compiled into a four-bit bytecode containing the opcode and the variable *id*(represents the order of marked variables). The bytecode will be inserted into the corresponding block. At runtime, the interpreter will execute the corresponding *opMark* and *opFree* functions according to the records in the jump Table The execution of *opFree* will add an entry to the memory pool maintained by OP-TEE, which contains a pointer to the released area and its size. This kind of memory recycling actually establishes the mapping of memory locations for the corresponding variable according to the timing of the memory store operation and releases them at the appropriate time. RMM can avoid explicit bugs such as double-free and releases of non-existing variables by ignoring them during execution and prompting the developer to fix them.

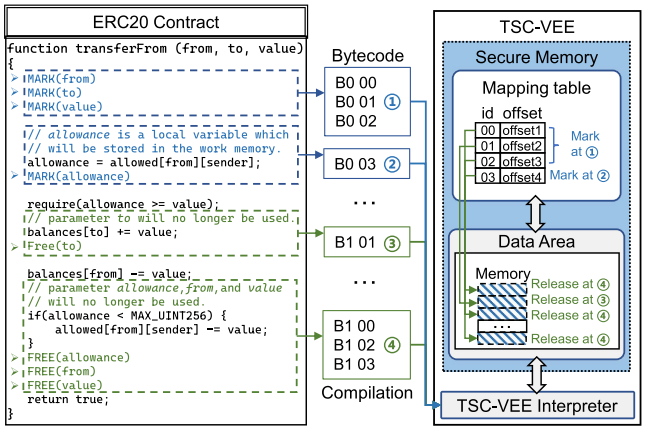


Fig. 5. The processes of memory recycling.

As shown in Fig. 5, we take the $transferFrom$ function in the typical ERC20 [44] contract (which is usually used as smart contract evaluation benchmark) as an example for analysis. This function transfers a specific amount of tokens from one account to another. The function has three input parameters: the transfer-out address $from$, the transfer-in address to, and the transfer amount $value$. This function also uses a local variable $allowance$ to represent the amount that the $from$ authorizes the $sender$ to use. These parameters will be stored in the work memory. When the balance of the transfer-in account decreases, the parameter to will no longer be used. Once the function logic is executed over, the parameters $allowance$, $from$, and $value$ will no longer be used. These parameters will be released together with the completion of the function. Taking into account the resource limitations of TrustZone, contracts with large work memory requirements can not be executed due to memory limitations. In TSC-VEE, developers can utilize the runtime memory management mechanism by calling the *MARK* and *FREE* APIs when writing the contract, as shown in Fig. 5. The mark will also be interpreted just like other instructions. TSC-VEE interpreter will map the parameters $from$, to, $value$ when executing bytecode block ① and map the parameter $allowance$ when executing bytecode block ②. The memory area of to will be released when executing bytecode block ③. Similarly, at bytecode block ④, the memory area of $allowance$, $from$, and $value$ will be released. This mechanism is closely related to the execution logic of the contract. When executing the function with large-scale parameters, the dynamic release can obviously reduce the runtime memory footprint.

E. Hybrid Granularity Bytecode Analysis

In addition, we designed a hybrid granularity bytecode analysis algorithm based on bytecode pre-execution. During contract execution, the detection of stack height and gas cost is performed to ensure execution correctness. Traditionally, this detection is performed before executing each instruction. Such frequent detection brings high latency. Inspired by the detection method

Algorithm 1. HYBRID GRANULARITY BYTECODE ANALYSIS

Input: Function bytecode BC, Instruction JumpTable JT
Output: the analysis results set AR, the max growth of the Stack height S_{mg}
/* $block$: record of basic block, S_c: stack change */
$AR \leftarrow null$, $S_{mg} \leftarrow 0$, $block \leftarrow null$, $S_c \leftarrow 0$
$begin \leftarrow 0$, $end \leftarrow BC.size() - 1$, $pos \leftarrow begin$
while $pos \neq end$ **do**
  $op \leftarrow BC[pos]$, $op\_info \leftarrow JT[op]$
  $pos \leftarrow pos + 1$, $block\_over \leftarrow false$
  S_c += $op\_info.sc$
  $S_{mg} \leftarrow Max(S_{mg}, S_c)$
  /* gc: gas cost. */
  $block.gc$ += $op\_info.gc$
  /* jo: jump offset, jt: jump target */
  if op is $op\_jumpdest$ **then**
    $AR.jo.append(pos - begin - 1)$
    $AR.jt.append(AR.instrs.size - 1)$
  else
    $AR.instrs.append(opcode\_info.fn)$
  end if
  $instr \leftarrow AR.instrs.get()$
  if op is the end instruction of a block **then**
    $block\_over \leftarrow true$
  else if op is any push instruction **then**
    $push\_end \leftarrow pos + op - op\_push1$
    $AR.push\_values.append(BC[pos : push\_end])$
    $pos \leftarrow push\_end + 1$
  else if op is instruction with dynamic gas cost **then**
    $instr.arg \leftarrow block.gc$
  end if
  if $block\_over$|| ($pos \neq end$ && $BC[pos]$ is $op\_jumpdest$) **then**
    $AR.instrs[block.begin].arg \leftarrow block.close()$
    $block \leftarrow newBlock()$
  end if
end while

of evmone at basic block granularity[4], HGBA analyzes the stack height and gas cost at different granularity before execution to reduce the number of detection and thus reduce the execution latency.

The analysis of stack height in HGBA is performed at the granularity of the contract. The change of stack height is caused by the execution order of instructions. According to the instruction interpretations, the change of stack height caused by each instruction is fixed. We take the ADD instruction as an example. This instruction pops out two operands from the stack and adds them, then pushes back the result to the stack. So in this process, the stack height will first decrease by two during execution and then increase by one. After execution, the stack height is reduced by one compared to the original height. Based on this

[4]https://github.com/ethereum/evmone/blob/master/docs/efficient_gas_calculation_algorithm.md

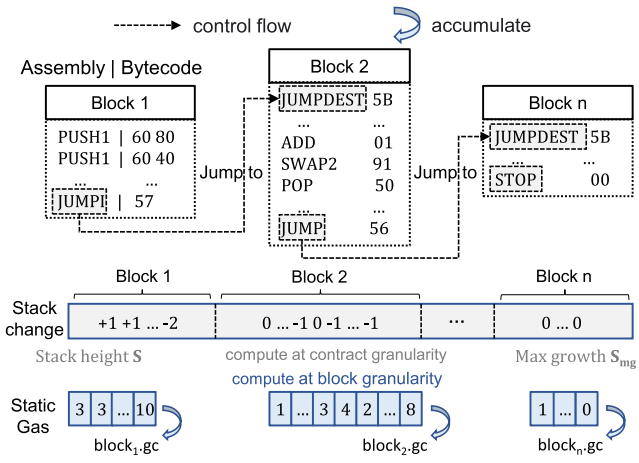


Fig. 6. Example of HGBA process.

principle, we can compute the actual stack height change and the maximum stack height change of each instruction. When the runtime stack height exceeds the maximum stack height, a stack overflow exception will occur. Stack overflow detection of HGBA is performed by pre-executing the bytecode before contract execution for a single time, rather than performed during the execution of each instruction as the traditional runtime detection does. HGBA can help reduce redundant computation by caching the maximum value of stack height. TSC-VEE will throw an exception and will not perform the execution when the stack overflow occurs.

The analysis of gas cost is performed at the granularity of the basic block. A basic block is a sequence of instructions that do not contain jumps. The gas cost of contract execution includes two parts. One is the static gas cost of the instructions, and the other is the dynamic gas cost that needs to be calculated according to the actual work memory consumption at runtime. HGBA performs static gas cost computation through bytecode pre-execution. The calculation results are recorded at the granularity of basic blocks. The dynamic gas cost will be added to the final result during execution. With HGBA, the number of gas cost detection can be reduced from once per instruction to once per basic block. In the analysis process, HGBA also completes the execution preparation to reduce the latency of the execution stage. The instruction table containing pointers to the instruction functions, the jump targets of each block, and parameters of several special types of instructions will be recorded and used as input data for the execution stage.

Algorithm.1 shows the process of the hybrid granularity bytecode analysis algorithm. HGBA fetches the opcodes from the bytecode according to the execution logic and decodes them into instructions. It calculates the stack height through the stack change S_c and the maximum growth of stack S_{mg} of each instruction and records the static gas cost in the basic block unit. The instruction table, jump offset, jump target, and instruction parameters will also be recorded as the execution preparation during analysis. We also give an example of HGBA in Fig. 6. The contract bytecode consists of serval basic blocks. During execution, the basic block jumps to another according to the control flow. The stack height is accumulated at the contract granularity, while the gas cost is accumulated at the block granularity. TSC-VEE can reduce the redundant computation by caching the analysis results of HGBA. Before performing HGBA to execute a contract function, TSC-VEE first looks up whether there is a record of this function in the mapping table or not. If the record exists, TSC-VEE will only perform the execution preparation and dynamic gas computation. For a new contract function, TSC-VEE performs HGBA in advance, computes and records the maximum height of the stack and the static gas cost, and completes execution preparation. This helps us avoid the overflow exception and reduce the redundant computation.

The analysis granularity of stack overflows and gas cost affects the times of runtime detection and the granularity of locating exceptions. When analyzing the bytecode at the basic block granularity, runtime detection needs to be performed for each basic block. And when an exception occurs, the developers can locate the basic block where the abnormal instruction locates. According to the frequency of stack overflow and gas shortage exceptions, we set the stack overflow detection at the contract granularity and gas cost computation at the basic block granularity to obtain better execution performance. When the exception occurs, the developers can verify the instructions in the abnormal part one by one to locate it. Therefore, HGBA achieves a balance between performance and reliability.

F. Cross-Isolation-Environment Prefetching

The execution of the smart contract function on TSC-VEE requires various types of data, including input parameters, contract bytecode, and blockchain state. All of the data can be obtained from transaction parameters or blockchain StateDB on the REE side. The blockchain StateDB stores the account states in the form of key-value. Each account address corresponds to an account storage trie, and the persistent storage data of the smart contract is stored in the trie corresponding to the smart contract address. During contract execution, the interpreter takes the offset from the stack top and obtains the index through cryptographic computation so as to accurately access the required data in the account storage trie. As mentioned in Section II-C, the cross-isolation-environment data fetching of the persistent storage data is a time-consuming operation that needs to go through complex processes. We propose the CIEP method to load all the required parameters to the TEE side at a single time.

CIEP is mainly oriented to the persistent storage data in the account storage trie. It prefetches all the data entries in the account storage trie as a superset of the reading and writing objects from the blockchain stateDB according to the contract address. The storage data will be stored with contract bytecode and function parameters in the shared memory to avoid increasing the secure memory footprint. During execution, the persistent storage data can be modified and flushed to the blockchain stateDB when necessary.

Fig. 7 shows the processes of the cross-world data load with and without CIEP. Without CIEP, this operation first switches from the secure world to the normal world, fetches and copies data into the shared memory via host application, then switches

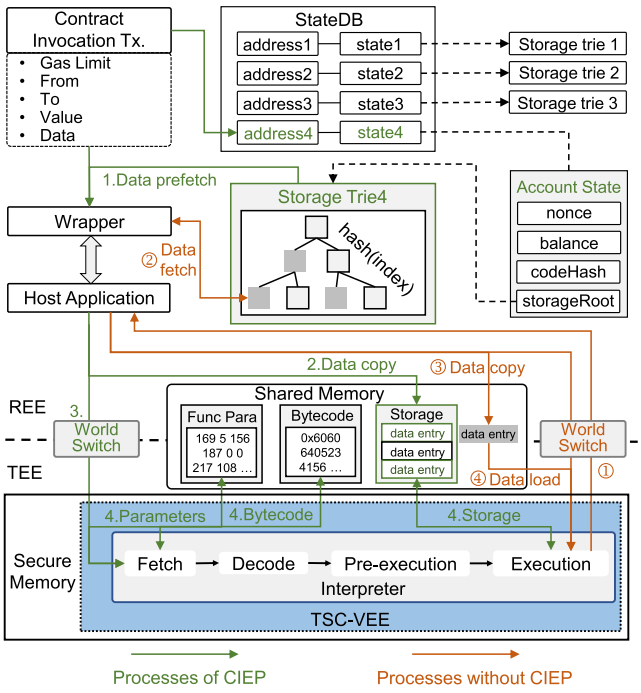


Fig. 7. The process of CIEP.

back to the secure world, loads data from shared memory, and continues to complete execution as ① ② ③ ④ show. The processes of cross-world data storage are similar but in reverse order. These time-consuming operations will be repeatedly executed at runtime once the interpreter encounters the SSTORE and SLOAD instructions.

As shown in Fig. 7, under the CIEP mechanism, the persistent storage data can be fetched with other parameters before execution rather than at runtime. The processes of CIEP are as follows: (1) The wrapper in the normal world will fetch the parameters from the contract invocation transaction and the account storage trie that contains the persistent storage in blockchain stateDB before execution. (2) All of the data will be copied to the shared memory between REE and TEE via the TSC-VEE host application. (3) TSC-VEE TA will be invoked through world switching. (4) With this data, the TSC-VEE interpreter can get the parameters, opcode, and storage data on-demand in different execution stages from the shared memory. A special case is that when an inter-contract call occurs, CIEP only prefetches the persistent storage data of the caller contract, since the blockchain state of the called contract cannot be obtained statically. A new contract execution process will be created to execute the called contract, and CIEP will be performed again as (1)–(4).

In general, data-prefetching is achieved through data location prediction and parallel data-fetching to cover the latency of data fetch in execution. The execution latency of mainstream ERC20 contracts is usually several hundred μs, and runtime data location prediction will bring high additional latency. In addition, under the existing mechanism of OP-TEE, multi-threading is not supported by TAs [25], which means that parallel executing and data-fetching is infeasible. The cross-world data fetching still needs to be completed by saving and switching the world state. In fact, CIEP prefetches a superset of the reading and writing objects from the contract level. There are two ways to narrow down this superset. One is to perform an accurate static analysis of data location during compiling. However, such static analysis requires language-level support. For example, variables should be assigned with constant addresses, which is not supported in Solidity. The other one is to record the positions of reading and writing objects accurately through contract pre-execution. This way is suitable for the scenario of transaction verification on the cloud. The miner records the reading and writing positions when packaging the transaction, and the verifier prefetches the data in advance for transaction verification. Introducing this pre-execution to TSC-VEE (which serves as an execution environment on the device) will bring a large extra latency. Overall, CIEP is a coarse-grained prefetching method that can maintain effectiveness while avoiding the extra latency of analysis.

IV. Performance Evaluation

In this section, we evaluate the performance of TSC-VEE. We intend to answer the following questions:

- What does TSC-VEE supporting the execution of the mainstream smart contracts looks like?
- What is the performance of executing smart contracts using TSC-VEE on the TEE compared with the REE?
- What is the performance improvement brought by the optimization mechanisms of TSC-VEE?
- How much performance improvement does TSC-VEE achieve in terms of memory footprint?

We will answer these questions through the following two aspects of evaluation: (1) the executing latency and memory footprint of TSC-VEE. (2) the performance improvement brought by our design.

A. Experimental Setup

Hardware and software. We used the Raspberry Pi 3B+ board as the experimental platform. The Raspberry Pi 3B+ embeds ARM TrustZone (ARMv8-A). It is equipped with the ARM Cortex A53 CPU (1.4 GHz, 4Cores) and 1 GB of memory. We use OP-TEE (version 3.8.0) as the trusted operating system. In the normal world, we use Linux-for-arm 4.14 system as the REE operating system to execute the client part and use other non-secure resources. To further deeply evaluate the performance of TSC-VEE, we also deploy Go-Ethereum[5](Geth, version 1.10.4), and evmone(version 0.8.2) on the REE side of the Raspberry Pi board and implement evmone on the TEE side. Geth is the most widely used Ethereum client. It provides a virtual machine named Ethereum Virtual Machine (EVM) for smart contract execution. The EVM is embedded in the Geth client. Evmone is a standalone EVM implementation that aims to provide fast and efficient execution of Solidity smart contracts, which is currently the fastest execution environment.

Metrics. TSC-VEE is the first execution environment on TrustZone for Solidity smart contracts. Due to the differences in hardware mechanisms and runtime environments, it is difficult

[5]https://github.com/ethereum/go-ethereum

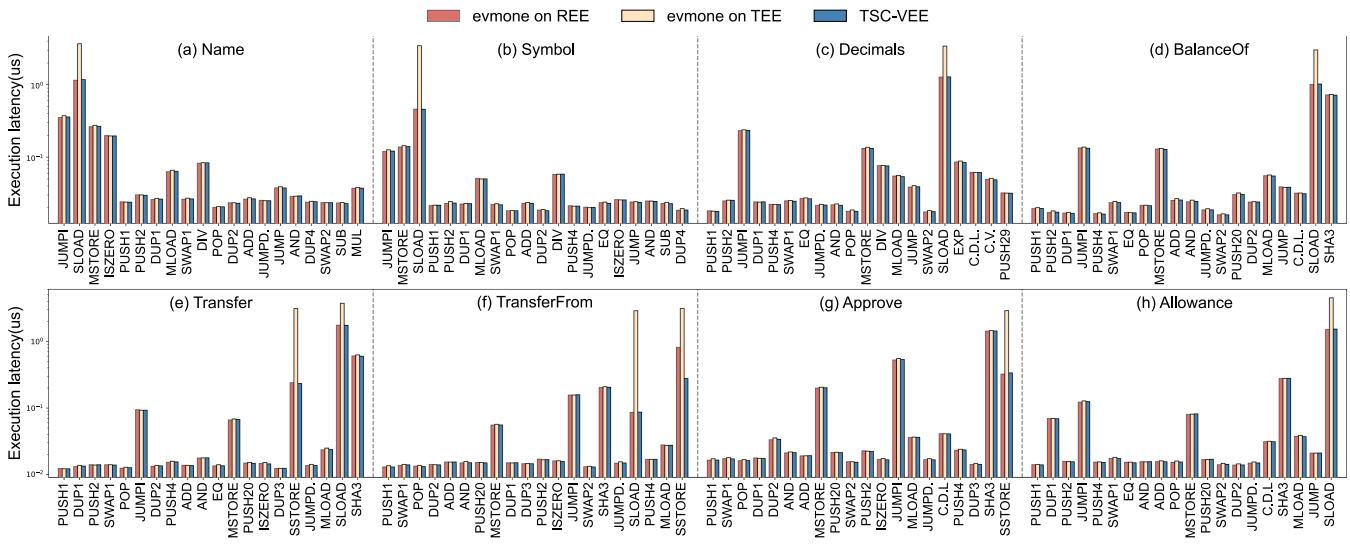


Fig. 8. The execution latency of TSC-VEE instructions.

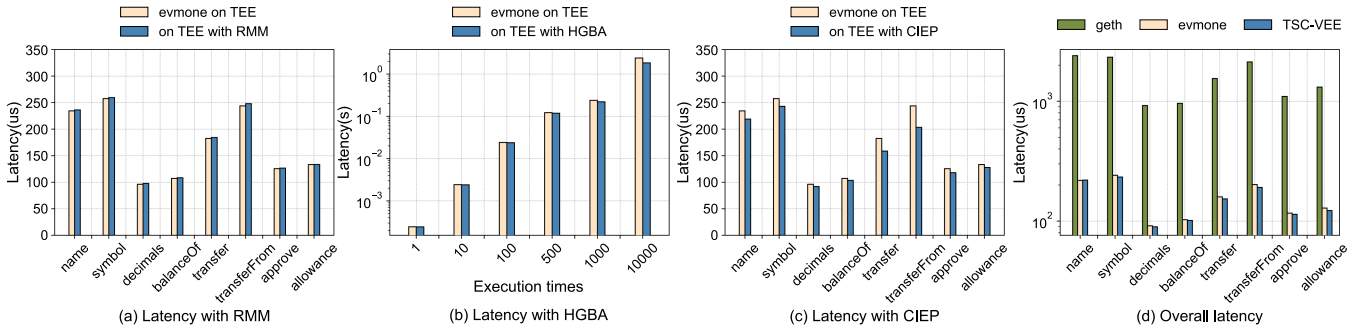


Fig. 9. The overall execution latency of TSC-VEE and the optimization mechanisms.

to make a fair comparison with other mechanisms on TrustZone or the existing execution environments on SGX for Solidity. So we perform evaluations on TSC-VEE, evmone, and Geth by executing the ERC20 contract [44], which is usually used as smart contract evaluation benchmark. The typical ERC20 contract provide eight functions including *name*, *symbol*, *decimals*, *balanceOf*, *transfer*, *transferFrom*, *approve*, and *allowance*. These functions are used for token creation, transfer, and so on, and have different blockchain state operations. The evaluation metrics are as follows:

- *Execution latency:* The time cost of the different stages during the execution of the ERC20 contract functions.
- *Memory footprint:* The memory consumption in the different stages during the execution of the ERC20 contract functions.

B. Execution Latency

We consider execution latency from two aspects: instruction latency and end-to-end latency. The instruction latency represents the time cost of executing an instruction. The end-to-end latency represents the overall time cost of all execution processes. For the convenience of comparison, we implement a C language version evmone and denote it as evmone on REE. We migrate the evmone in C language to the TEE side directly following the TA API without any optimization mechanism, and denote it as evmone on TEE. Results of execution latency are shown in Figs. 8 and 9, and Table III.

TABLE III
LATENCY OF OPTIMIZATION TECHNIQUES IN TSC-VEE DURING EXECUTING ERC20 FUNCTIONS

Function	Execution (μs)	RMM (μs)	HGBA (μs)	CIEP (μs)	Others (μs)
Name	219.80	1.76	20.14	8.99	188.90
Symbol	233.09	1.87	24.52	9.51	197.19
Decimals	89.71	1.47	7.87	9.18	71.20
BalanceOf	101.36	1.07	8.92	9.49	81.88
Transfer	153.12	1.72	17.43	9.71	124.26
TransferFrom	191.07	4.19	29.12	10.02	147.75
Approve	113.93	1.13	12.02	9.85	90.93
Allowance	122.52	0.23	13.43	9.89	98.97

1) Instruction Latency: First of all, we test the instruction latency and verify the effect of executing the eight functions

in the mainstream ERC20 contract. The function *name*, *symbol*, and *decimals* just get data from the blockchain storage without any parameter, since function *balanceOf*, *allowance* get data according to the parameter. Function *approve*, *transfer*, *transferFrom* modified the data store in the blockchain stateDB. The computational complexity of these three types of functions is increasing. These functions of the ERC20 contract involve 43 instructions in our instruction sets. We execute the above functions 1000 times on evmone(on REE), evmone(on TEE), and TSC-VEE, respectively. The average results of the 20 most frequently used instructions of each function are shown in Fig. 8. According to the results, the execution latency of instructions related to CIEP can be greatly reduced. For example, the latency of SLOAD has been reduced to the same level as the evmone(on REE) by avoiding world switching. Moreover, our optimization mechanisms do not increase the computational load of the instructions. The execution latency of other instructions on REE and TEE sides is almost the same, the differences between them are from -2.03% to 2.21%. This is because the REE side and the TEE side are actually two different states of the same CPU core, which are switched according to the time slices. There is no essential difference in computing performance between them. The time-consuming instructions include SLOAD, MSTORE, SHA3, DIV, and JUMPI. The SLOAD and MSTORE instructions involve time-consuming operations, including data access and copy. The SHA3 and DIV perform division and cryptographic calculations with high computational complexity. Besides determining whether to jump, JUMPI also needs to verify that the jump address exists and that the jump target is a JUMPDEST instruction. The verification takes extra time. In addition, the results also verify the correctness of the instructions in TSC-VEE.

2) End-to-End Latency: We further consider the end-to-end latency from two aspects: the latency change caused by TSC-VEE design and the overall latency. The latency change caused by design aims to evaluate the performance improvement brought by RMM, CIEP, and HGBA of TSC-VEE. Overall latency represents the time from the initiation of the contract execution request to the completion of the execution. For Geth and evmone, the overall latency is the real execution latency. For TSC-VEE, the overall latency contains fourfold: the latency of CIEP, the latency of HGBA, the latency of RMM, and the latency of instruction execution.

We test the end-to-end latency by executing eight functions of the standard ERC20 contract 1000 times. On the basis of on TEE (evmone on the TEE side without optimization mechanisms), we add the RMM, HGBA, and CIEP mechanisms to evaluate the change in overall execution latency. The latency change caused by RMM is shown in Fig. 9(a). According to the results, RMM can help reduce the memory footprint with only 0.97% additional latency overhead. This is because using RMM will increase the number of instructions. Each mark added by developers while programming will be compiled into an extra instruction in the bytecode. This method increases the latency for memory release at the instruction level. It should be noted that the out-of-memory cases do not occur in the above test. The extra latency of such cases and improvement in memory footprint will be discussed in detail in Section IV-C.

Fig. 9(b) shows the performance improvement brought by the hybrid granularity bytecode analysis during the execution of *transferFrom*. When executing the function only once, the execution latency of HGBA is slightly shorter than that of traditional runtime stack overflow detection and gas cost computation. This is because that HGBA reduces the times of stack height and gas remaining detection. With the increase in execution times, the performance improvement brought by the HGBA method becomes more obvious. This is because on TEE with HGBA does not need to perform the HGBA method during repeated execution. Without the caching mechanism of HGBA, the detection should be performed at runtime every time we execute the function. As a result, this method can bring an average of 6.04% latency reduction, with a maximum of 23.49%.

The latency reduction caused by CIEP is shown in Fig. 9(c). Results show that CIEP reduces the execution latency by 7.48% on average. This latency is related to the size of the prefetched data and the time of persistent storage operations. For function *name* without input parameter, this latency is 8.99us. For function *transferFrom* with the most input parameters, this latency is 10.02us. Without the CIEP method, all the persistent storage(SSTORE and SLOAD) instructions during the execution process need to penetrate the execution environment. This process includes world switching and data copy via the shared memory. According to the results, the cross-world data access latency accounts for 10.67% of the execution latency. The average execution latency of the cross-world data access instruction is 4.97us.

The overall execution latency results on evmone(on REE), TSC-VEE, and Geth are shown in Fig. 9(d). Through the optimization mechanisms, TSC-VEE has achieved an average performance improvement of 12.63%. And its execution performance is 3.79% faster than evmone(on REE) on average. Compared with the Geth client, TSC-VEE has a $9.29\times$ performance improvement. This improvement comes from several aspects. First, TSC-VEE achieves $1.13\times$ performance improvement through the optimization mechanisms. Second, TSC-VEE adopts the mode of analyzing the instructions first, preparing parameters for execution, and then executing, while Geth executes the instructions one by one directly. Third, TSC-VEE is an independent module while the EVM of Geth is embedded in the client. There is a certain amount of client overhead during the measuring. In addition, TSC-VEE is implemented in C language while Geth is implemented in Go language. The difference in the programming language will also bring a certain performance gap. Overall, these differences resulted in a performance improvement of about $8.22\times$.

In addition, we evaluate the execution latency of each optimization mechanism itself during the execution of TSC-VEE. The results are shown in Table III. The total execution latency in TSC-VEE includes the latency of RMM, HGBA, CIEP, and the others. These four parts account for 1.10%, 7.01%, 10.67%, and 81.22% of the total execution latency, respectively. It is worth noting that in the above mechanism, only RMM introduces

TABLE IV
MEMORY FOOTPRINT OF EACH PART IN TSC-VEE DURING EXECUTING ERC20 FUNCTIONS

Function	Total (KB)	Total of TSC-VEE (KB)	Execution (KB)	Total of CIEP (KB)	Input (KB)	Bytecode (KB)	T.X. (KB)	Key (KB)	Storage (KB)	TA (KB)	Percentage (%)
Name	1226.85	437.81	302.52	8.53	0.00	6.91	0.28	0.73	0.61	198.23	35.69
Symbol	1010.97	444.42	309.14	8.53	0.00						43.96
Decimals	1094.40	426.93	**291.64**	8.53	**0.00**						39.01
BalanceOf	1804.38	692.24	556.74	8.63	0.11						38.36
Transfer	2191.07	798.24	662.61	8.70	0.17						36.43
TransferFrom	2183.98	1044.36	**908.52**	8.80	**0.28**						47.82
Approve	1759.92	738.75	603.13	8.70	0.17						41.98
Allowance	1892.39	727.30	591.59	8.74	0.21						38.42

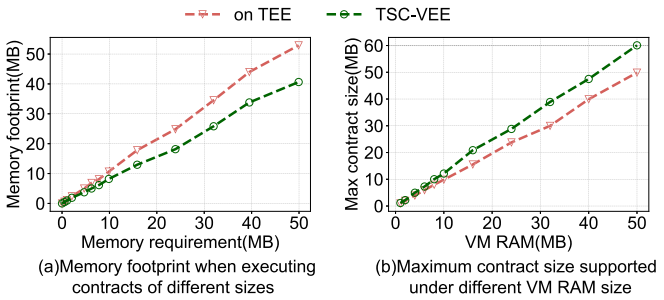


Fig. 10. The performance of TSC-VEE and on TEE for contracts with different memory requirements.

additional latency overhead. HGBA and CIEP execute the steps in the critical path in advance, rather than introducing new execution steps. It implies that these three mechanisms can bring large performance improvements as mentioned above, only with negligible latency overhead.

C. Memory Footprint

We consider memory footprint from two aspects: overall memory footprint and work memory footprint. The overall memory footprint represents the memory consumption of all execution processes of TSC-VEE. The work memory footprint refers to the consumption of TSC-VEE's work memory during execution, which is used to evaluate the effect of the runtime memory management mechanism. Results of memory footprint are shown in Table IV and Figs. 10, 11, and 12.

1) Overall Memory Footprint: We first pay attention to the memory performance of TSC-VEE in terms of overall memory footprint. Table IV shows the memory footprint of each part in TSC-VEE when executing the ERC20 functions. The overall memory footprint contains the consumption of TSC-VEE and the consumption of OP-TEE OS. The memory footprint of TSC-VEE accounts for 40.21% of the overall memory footprint on average, including the TA binary file, the prefetched data, and runtime consumption. The runtime consumption is positively correlated with the computational complexity of the function, and the consumption ranges from the least 291.64 KB to the most 908.52 KB. The prefetched data contains the smart contract bytecode, the transaction parameters, the blockchain state, the RSA public key, and the function parameters. For different functions in the ERC20 contract, the difference in the prefetched data lies in the size of the function parameters. The size of the prefetched data (CIEP) is related to the logic of the smart contract. The more persistent storage data (essentially the global variable) involved in the contract, the larger the size of the prefetched data will be. We tested different functions of the same ERC20 contract. These functions prefetch different data entries from the same area and the number of data entries pre-fetched is similar. At the same time, the size of each data entry is fixed at 256 bits, which is relatively small compared to all the pre-fetched data. Therefore, the size of prefetched data is similar.

2) Work Memory Footprint: We evaluate the performance improvement brought by the runtime memory management mechanism of TSC-VEE when executing different sizes of contracts (contracts with different memory requirements). We use the *transferFrom* function mentioned in Section III-D to simulate contracts in different sizes through loop execution. Our mechanism will release the three parameters after executing the relevant code block. To simulate devices with rich resources, we increase the size of the secure memory on the Raspberry Pi 3B+ to 64 MB by modifying the value of *PGT_CACHE_SIZE* (the number of page tables in virtual memory) and *CFG_TZDRAM_SIZE* (the secure memory size managed by OP-TEE OS). We first evaluate the memory footprint of TSC-VEE and on TEE when executing contracts of the same size. As shown in Fig. 10(a), the memory footprint of TSC-VEE is lower than that on TEE, with an average of 23.50%. Then, we evaluate the maximum contract size supported on TSC-VEE and on TEE under different work memory sizes. The size of the secure memory for OP-TEE set on the Raspberry Pi 3B+ is 16 MB, which is divided into three parts: TEE RAM, TA RAM, and VM_RAM. We adjust the size of the work memory available for the execution environment by setting the size of VM_RAM. The results are shown in Fig. 10(b). The experimental results show that our memory recycle mechanism can help to run contracts of larger size under the same VM_RAM size compared with on TEE. The difference between the two lines in the figure represents the part of the work memory released by our memory recycle mechanism. This improvement is 22.95% on average.

We evaluate the performance of RMM when the size of work memory cannot meet the memory requirement of contract. As mentioned in Section III-D, in this case, the execution environment will write out a certain part of the data from the work memory to the REE side. During this process, TSC-VEE moves the data to the shared memory, and switches to the REE side. The position of the data in shared memory will also be passed as a parameter to the host application. Then the host application

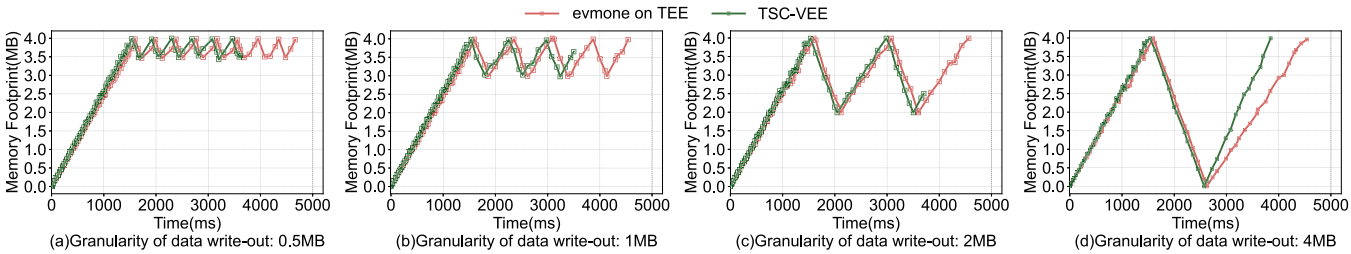


Fig. 11. The change of memory footprint of TSC-VEE and on TEE over time.

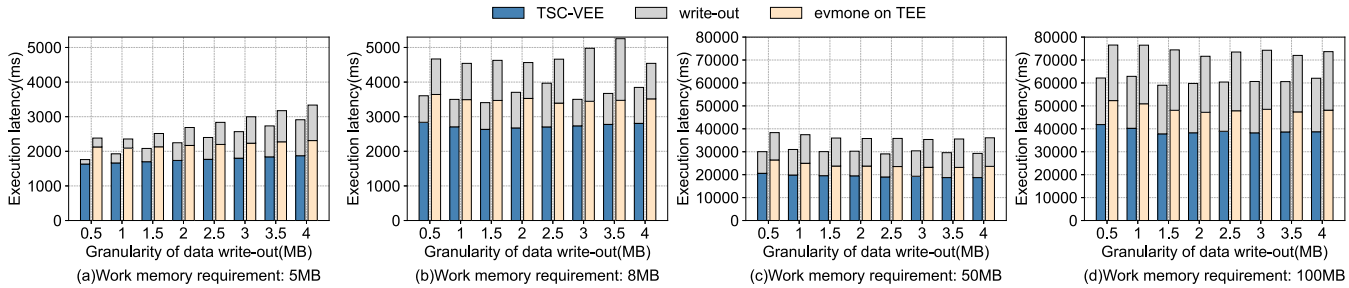


Fig. 12. The execution latency of TSC-VEE and on TEE for contracts with large memory requirements.

moves the data from the shared memory to other memory areas, and switches back to the TEE side to continue executing. Therefore, a write-out operation only involves world state switches twice. We add this write-out mechanism for on TEE to compare with TSC-VEE. We set the VM_RAM to 4 MB and execute the contract with an 8 MB memory requirement using TSC-VEE and on TEE, respectively. Fig. 11 shows the change of memory footprint over time under different write-out granularity. According to the write-out granularity, the execution environment needs to perform data write-out several times to meet the requirements of contracts Since TSC-VEE performs memory recycling at runtime, its memory footprint is lower. Under the same work memory size, the size of memory that TSC-VEE needs to obtain by data write-out is smaller. Therefore, the times of data write-out of TSC-VEE are less than on TEE, as shown in Fig. 11(a) and (b). Fewer data write-out times make the execution latency of TSC-VEE lower. These reasons make TSC-VEE performs much faster than on TEE. Also, according to the discussion in Section IV-B, we know that TSC-VEE performs faster than on TEE by about 12.63%. This implies that even the memory footprint of TSC-VEE and on TEE seems to be similar at the same time, TSC-VEE has actually satisfied more memory requirements.

We further evaluate the execution latency under large memory requirements in detail, and the results are shown in Fig. 12. The size of VM_RAM is set to be 4 MB. Fig. 12(a), (b), (c), and (d) shows the execution latency of TSC-VEE and on TEE with different write-out granularity under the memory requirement of 5, 8, 50, and 100 MB, respectively. Overall, due to differences in data write-out times and execution speed, the execution performance of TSC-VEE is $12.71\% \sim 30.11\%$ faster than on TEE. According to the results, we can find the following phenomena:

1) The more data we write out, the greater the write-out latency is.
2) When a lot of data has been written out, the execution latency will increase slightly, since several data on REE side need to be accessed during execution.
3) If the write-out granularity does not match the memory requirement, there may be cases where only 0.1 MB of memory is required, but 4 MB of data wrote out. This case will increase the times of data write-out and significantly increase the execution latency.

So we can observe that the size of write-out data should be as close as possible to the real memory requirement to achieve faster execution. From this perspective, smaller write-out granularity can make the size of data written out closer to the memory requirement. When writing out data of the same size, small-grained write-out will increase the times of write-out. Each write-out operation means switching twice between the TEE side and the REE side. Compared with the latency of data movement, this µs-level switch latency is almost negligible. Our experimental results also show that the execution latency is generally close to optimal at a small granularity of 0.5 MB. This conclusion can provide a reference for setting the write-out granularity.

D. Security and Scalability Discussion

Security Analysis. TSC-VEE follows the standard workflow of the confidential smart contract and focuses on the contract computation stage. Before execution, the user encrypts the blockchain state and contract parameters and stores it in the shared memory as the input. TSC-VEE decrypts the data and then executes the smart contract. TSC-VEE prefetches the persistent storage data to the TEE side with CIEP to avoid privacy leakage during runtime data interaction. Relying on TrustZone's hardware-enforced isolation, TSC-VEE completes the contract execution process and generates the new blockchain state. The

results are also encrypted and stored in shared memory and used after decryption. Thus, TSC-VEE maintains the same security property as the existing confidential smart contract under our threat model, but cannot deal with the rollback attacks caused by non-deterministic consensus protocols.

Scalability Analysis. TSC-VEE is deployed on the node of the blockchain system as the smart contract execution environment like EVM. With the delicately designed optimization mechanism, TSC-VEE can achieve similar execution performance to that in non-secure environments, without significant additional overhead at the single point. From the perspective of blockchain architecture, TSC-VEE operates on a different level than the consensus and network mechanisms. It does not affect the workflow of smart contracts and the scalability of the blockchain system. It can be easily extended to the blockchain network by equipping the node with TrustZone like previous work does [29].

V. Conclusion

In this article, we propose TSC-VEE, the first virtual execution environment to support mainstream smart contracts programmed by Solidity Language running on TrustZone. We first design a specific instruction set for Solidity smart contracts adapted to the execution mechanism of TrustZone. TSC-VEE has achieved competitive performance by three proposed optimization technologies, the runtime memory management mechanism, the hybrid granularity bytecode analysis algorithm, and the cross-isolation-environment prefetching. Experimental results illustrate that TSC-VEE can support mainstream Solidity contracts performing on TrustZone with competitive execution efficiency and memory footprint. The source code TSC-VEE can be found at https://github.com/nkicsl/TSC-VEE.

References

[1] 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.

[2] W. Zou et al., "Smart contract development: Challenges and opportunities," *IEEE Trans. Softw. Eng.*, vol. 47, no. 10, pp. 2084–2106, Oct. 2021.

[3] T. Lu and L. Peng, "BPU: A blockchain processing unit for accelerated smart contract execution," in *Proc. IEEE/ACM 57th Des. Automat. Conf.*, 2020, pp. 1–6.

[4] X. Guo, Q. Guo, M. Liu, Y. Wang, Y. Ma, and B. Yang, "A certificateless consortium blockchain for IoTs," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst.*, 2020, pp. 496–506.

[5] O. Novo, "Scalable access management in IoT using blockchain: A performance evaluation," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4694–4701, Jun. 2018.

[6] Y. Du, H. Duan, A. Zhou, C. Wang, M. H. Au, and Q. Wang, "Towards privacy-assured and lightweight on-chain auditing of decentralized storage," in *Proc. IEEE 40th Int. Conf. Distrib. Comput. Syst.*, 2020.

[7] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (SOK)," in *Proc. Int. Conf. Princ. Secur. Trust*, 2017, pp. 164–186.

[8] R. Li, Q. Wang, Q. Wang, D. Galindo, and M. Ryan, "SOK: Tee-assisted confidential smart contract," *Proc. Privacy Enhancing Technol.*, vol. 2022, pp. 711–731, Aug. 2022.

[9] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.

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

[11] A. M. Devices, "Secure encrypted virtualization API: Technical preview," in *Proc. Adv. Micro Devices*, 2019, Art. no. 55766.

[12] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.

[13] E. Feng et al., "Scalable memory protection in the PENGLAI enclave," in *Proc. 15th USENIX Symp. Operating Syst. Des. Implementation*, 2021, pp. 275–294.

[14] 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. 2018.

[15] 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. 2021.

[16] Statista, "Arm's market share and targets across key technology markets in 2019 and 2028 fiscal years," 2022. [Online]. Available: https://www.statista.com/statistics/1132112/arm-market-share-targets/#statisticContainer

[17] 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. IEEE Int. Symp. Reliable Distrib. Syst.*, 2020, pp. 31–40.

[18] J. Lind, O. Naor, I. Eyal, F. Kelbert, E. G. Sirer, and P. Pietzuch, "Teechain: A secure payment network with asynchronous blockchain access," in *Proc. 27th ACM Symp. Operating Syst. Princ.*, 2019, pp. 63–79.

[19] 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.*, 2020, pp. 442–453.

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

[21] W. Gavin et al., "The solidity contract-oriented programming language," 2014. [Online]. Available: https://github.com/ethereum/solidity

[22] Z. Jian, "Analysis of popular smart contracts on ethereum," 2022. [Online]. Available: https://github.com/JolyonJian/contracts

[23] "Contract internal transactions & contracts with verified source codes only," 2021. [Online]. Available: https://etherscan.io

[24] K. Śliwak et al., "Internal of solidity, layout in memory," 2021. [Online]. Available: https://docs.soliditylang.org/en/v0.8.11

[25] W. Gavin et al., "The official documentations of OP-TEE," 2019. [Online]. Available: http://optee.readthedocs.io

[26] N. Szabo, "Formalizing and securing relationships on public networks," First monday, 1997, vol. 2, no. 9, .

[27] 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. Prog. Comprehension*, 2019, pp. 260–270.

[28] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[29] M. Russinovich et al., "CCF: A framework for building confidential verifiable replicated services," Technical report, Microsoft Research and Microsoft Azure, 2019.

[30] Y. Yan et al., "Confidentiality support over financial grade consortium blockchain," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2020, pp. 2227–2240.

[31] M. Brandenburger, C. Cachin, R. Kapitza, and A. Sorniotti, "Trusted computing meets blockchain: Rollback attacks and a solution for hyperledger fabric," in *Proc. IEEE 38th Symp. Reliable Distrib. Syst.*, 2019, pp. 324–32409.

[32] 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*, 2016, pp. 839–858.

[33] 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, 2018.

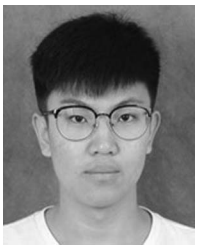
[34] R. Cheng et al., "Ekiden: A platform for confidentiality-preserving, trustworthy, and performant smart contracts," in *Proc. IEEE Eur. Symp. Secur. Privacy*, 2019, pp. 185–200.

[35] P. Das et al., "FastKitten: Practical smart contracts on bitcoin," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 801–818.

[36] H. Yin, S. Zhou, and J. Jiang, "Phala network: A confidential smart contract network based on polkadot," 2019. [Online]. Available: https://files.phala.network/phala-paper.pdf

[37] E. A. et al., "Enclave EVM(EEVM), an open-source, standalone, embeddable, C implementation of the ethereum virtual machine," 2019. [Online]. Available: https://github.com/microsoft/eEVM

[38] F. McKeen et al., "Intel software guard extensions (intel SGX) support for dynamic memory management inside an enclave," in *Proc. Hardware Architectural Support Secur. Privacy*2016, pp. 1–9.
[39] X. Li et al., "Design and verification of the ARM confidential compute architecture," in *Proc. 16th USENIX Symp. Operating Syst. Des. Implementation*, 2022, pp. 465–484.
[40] N. Santos et al. , "Using arm trustzone to build a trusted language runtime for mobile applications," in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2014, pp. 67–80.
[41] K. Ryan, "Hardware-backed heist: Extracting ECDSA keys from qualcomm's trustzone," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 181–194.
[42] 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.*, 2019, pp. 195–209.
[43] J. F. et al., "Arm speculation barrier," 2017. [Online]. Available: https://github.com/ARM-software/speculation-barrier
[44] V. Buterin and F. Vogelsteller, "ERC-20 token standard," 2015. [Online]. Available: https://theethereum.wiki/w/index.php/ERC20TokenStandard



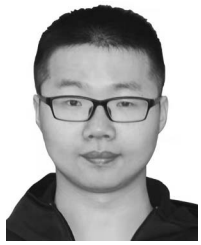
Zhaolong Jian received the BS degree in the Internet of Things from Nankai University, in 2020. He is currently working toward the PhD degree in the College of Computer Science, Nankai University. His main research interests include blockchain, smart contract, and Internet of Things.

Ye Lu received the BS and PhD degree from Nankai University, Tianjin, China, in 2010 and 2015, respectively. He is an associate professor with the College of Cyber Science, Nankai University now. His main research interests include FPGA accelerator, blockchain virtual machine, embedded system, Internet of Things.

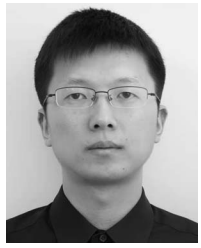
Youyang Qiao received the BS degree in Internet of Things from Nankai University, in 2020. She is currently working toward the the master's degree in computer science with Nankai University. Her main research is WebAssembly in the blockchain system.

Yaozheng Fang received the BS degree from the Hebei University of Technology, Tianjin, China, in 2019. He is currently working toward the PhD degree in the College of Computer Science, Nankai University. His main research interests include blockchain, smart contract and Internet of Things.

Xueshuo Xie received the PhD degree in engineering from Nankai University, in 2021, the BS and MA, SC degrees from Shandong University, Jinan, China, in 2011 and 2014. He is a postdoctoral in the College of Computer Science, Nankai University. He is currently working with the Intelligent Computing System Lab, College of CyberScience, Nankai University, Tianjin, China. His current research interests include IoT security, data-driven anomaly detection, and blockchain.



Dayi Yang received the MS degree from Fudan University, in 2017. He is now working with Antgroup as a software engineer. His main research interests include collaborative computing, blockchain technology and fintech.



Zhiyuan Zhou received the MS degree from Chinese Academy of Sciences, in 2007. He is now working with Antgroup as a software architect. His main research interests include cloud computing, blockchain technology and fintech.

Tao Li (Member, IEEE) received the PhD degree in computer science from Nankai University, China in 2007. He works with the College of Computer Science, Nankai University as a professor. He is the Member of the ACM, and the distinguished member of the CCF. His main research interests include heterogeneous computing, machine learning and Internet of things.