The ISC Int'l Journal of Information Security

ISeCure

January 2024, Volume 16, Number 1 (pp. 37–53)

http://www.isecure-journal.org

Safety Guards for Ethereum Smart Contracts

Morteza Amirmohseni [1] and Sadegh Dorri Nogoorani [1,*]

[1] *Blockchain Laboratory, Faculty of Electrical and Computer Engineering, Tarbiat Modares University, Tehran, Iran*

ARTICLE INFO.

Article history:
Received: October 17, 2022
Revised: May 10, 2023
Accepted: May 22, 2023
Published Online: November 20, 2023

Keywords:
Blockchain, Runtime Monitoring, Smart Contract, Vulnerability

Type: Research Article

doi: 10.22042/isecure.2023.365808.846

dor: 20.1001.1.20082045.2024.16.1.3.7

ABSTRACT

Smart contracts are applications that are deployed on a blockchain and can be executed through transactions. The code and the state of the smart contracts are persisted on the ledger, and their execution is validated by all blockchain nodes. Smart contracts often hold and manage amounts of cryptocurrency. Therefore, their code should be secured against attacks. Smart contracts can be secured either by fixing their source/byte code before deployment (offline) or by inserting some protection code into the runtime (online). On the one hand, the offline methods do not have enough data for effective protection, and on the other hand, the existing online methods are too costly. In this paper, we propose an online method to complement the offline methods with a low overhead. Our protections are categorized into multiple *safety guards*. These guards are implemented in the blockchain nodes (clients), and require some parameters to be set in the constructor to be activated. After deployment, the configured guards protect the contract and revert suspicious transactions. We have implemented our proposed safety guards with small changes to the Hyperledger Besu Ethereum client. Our evaluations show that our implementation is effective in preventing the corresponding attacks, and has low execution overhead.

1 Introduction

Ethereum is the world's leading public blockchain for smart contracts and decentralized application (DApp) development. It has revolutionized applications of blockchain and created a new era for blockchain technology. Smart contracts are small programs which are stored on-chain, and are triggered by special transactions (i.e. method calls). A contract verifies each request's conditions and performs appropriate action(s) automatically and without third party intervention. Smart contracts are unstoppable, and are executed exactly as programmed without manipulation or censorship.

Ethereum smart contracts are written in a Turing-complete language and can implement any algorithms. Since contracts usually hold and facilitate the exchange of cryptocurrency, they can be used to keep record of digital assets and stocks. Smart contracts are also used in a wide variety of other applications such as supply chain management, Internet of Things (IoT), digital identity management, electronic health record (EHR) systems, electronic voting, etc.

One of the most important issues with the blockchain technology is the security of smart contracts. Smart contracts are often used for financial purposes, and once deployed it is impossible to change

* Corresponding author.
Email addresses: `m.amirmohseni@modares.ac.im`, `dorri@modares.ac.i`

ISSN: 2008-2045 © 2024 ISC. All rights reserved.

their code even for bug fixes [1]. Therefore, their code should be secured against vulnerabilities and attacks. Smart contract security methods can be divided into two categories: offline and online. Offline methods take the source or byte code of a contract and report its vulnerabilities to the developer for corrections. Online methods insert the protection code into the source code, the byte code, or the runtime code.

Offline methods cannot detect all vulnerabilities due to the lack of access to real-time execution information. Online methods are also limited and have high overhead. Adding new code to a contract increases its gas consumption. In addition, it may not be possible for big contracts. For example, the Ethereum virtual machine (EVM) bytecode which corresponds to a smart contract is limited to be up to about 25KBs.

In this paper, we introduce an online method that complements the available offline methods, can protect the contract from significant attacks, and does not have the mentioned limitations of other online methods. By using our method, the contract owner can define customizable safety guards to prevent threats which may be caused even by unknown vulnerabilities. The guards are implemented in the blockchain runtime, and are enabled by inserting short code snippets (guards parameters) into the contract constructor. Our contributions are as follows:

- Safety guards are new contract protection methods which can be customized for each contract by its owner.
- Our method is not limited to known vulnerabilities and can also prevent some attacks which exploit zero-day vulnerabilities.
- Only a short code snippet is required to use a guard. The code is generated by a simple user interface (UI) and can be added to the contract with a minimum programming knowledge.

We present the related background material in Section 2; survey the related work in Section 3; explain our proposed method in Section 4; discuss our implementation in Section 5; perform a case study on each guard in Section 6; evaluate our method and compare it with the previous work in Section 7; and conclude the paper in Section 8.

2 Background

2.1 Blockchain and Smart Contracts

Blockchain is a trusted distributed ledger that shares data in a peer-to-peer network. This technology was first used by the Bitcoin network for the exchange of cryptocurrencies without the need for a trusted third party. As the name implies, blockchain is an ordered chain of blocks. Each block is identified by a hash value and is referenced by its successive block [1].

Blockchain applications are not limited to financial cases. It is possible to create programs called smart contracts that involve complex transactions and run automatically. The main idea is to embed contractual concepts in computer components. They provide better observability, verifiability, and enforceability. The parties of a contract can be informed about each others' performance, and are not concerned about correct execution of the code. Smart contracts can protect the privacy of the parties to some extent, and keep/publish as much information as needed [2].

Ethereum is the first blockchain which supports general-purpose smart contracts. The Ethereum virtual machine (EVM) is the runtime environment for the Ethereum smart contracts, and each network node executes an implementation of EVM [3]. Solidity is the most popular high-level programming language to write smart contracts for the EVM [4]. The source code is compiled into EVM bytecode and is deployed (stored) on the blockchain. Once deployed, the contract can be accessed by an address, has a dedicated storage, and can receive, keep, and transfer cryptocurrency, All Ethereum miners execute the contract's code, and record state changes on the blockchain [3]. When smart contract variables are set, their values are written to *storage slots*. Each Ethereum smart contract has a dedicated storage array of 32-bytes slots, initially set to zeros. The total number of storage slots is 2^{256}. Each slot is identified by a number starting from zero [5].

Ether is the native cryptocurrency of the Ethereum blockchain. Users send transactions to the Ethereum network for three purposes: (1) deploying new contracts, (2) invoking a contract's function, and (3) transferring Ether to users or contracts [6]. The history of all transactions is stored on the Ethereum public blockchain which is an append-only data structure. Each contract can hold an amount of Ether and has a dedicated storage. The sequence of transactions on the blockchain determines the balance and the state of the contracts [3].

2.2 Ethereum Clients

The Ethereum network consists of a distributed set of miner nodes that validate transactions, execute smart contracts, and store blocks. A fee is paid to the miners for their service which is measured in terms of *gas*. Gas can be considered as the fuel required to run the Ethereum network. It is the fee payed for transaction validation, block mining, and smart con-

[1] In many public blockchains including the Ethereum.

tract execution. Gas payment also acts as an inhibitor against wasting network resources by low-value and useless transactions. The *gas price* (in Ether) dynamically changes in a competitive manner according to a supply and demand mechanism [3].

A node runs an Ethereum client. The client implements the Ethereum protocols that verify transactions, create blocks, keep its data accurate, and secure the network. The Ethereum community maintains multiple open-source clients, developed by different teams using different programming languages [7]. Table 1 summarizes the specifications of these different clients.

3 Related Works

In this section, we introduce the researches in the field of smart contract security focusing on the protection of Ethereum smart contracts. Our review of the related works is categorized into offline and online methods which are presented subsequently.

3.1 Offline Methods

Luu *et al.* [14] investigated the security of Ethereum smart contracts, and highlighted four security weaknesses which may be exploited by an adversary to manipulate the execution of a smart contract and gain profit: (1) transaction-ordering dependence, (2) timestamp dependence, (3) mishandled exceptions, and (4) reentrancy. They developed the Oyente tool for symbolic execution of Ethereum smart contracts to detect these vulnerabilities. Among the 19,366 Ethereum smart contracts under their review, the Oyente flags 8,833 of them as vulnerable. TheDAO vulnerability which led to the loss of 60 million USD in June 2016, was also identified by Oyente.

Tsankov *et al.* [15] present Securify, a security analyzer for Ethereum smart contracts that is scalable, fully automated, and able to prove contract behaviors as safe/unsafe with respect to a given property. The input to Securify is the EVM bytecode of a contract and a set of security patterns. Securify's analysis consists of two steps. First, it symbolically analyzes the contract's dependency graph to extract precise semantic information from the code. Then, it checks compliance and violation patterns that capture sufficient conditions for proving if a property holds or not. To enable extensibility, all patterns are specified in a designated domain-specific language (DSL).

MythX [16] automatically scans for security vulnerabilities in smart contracts written for EVM-compatible blockchains. When developers submit their code to the MythX API, it gets analyzed by multiple microservices in parallel: A static analyzer that parses the abstract syntax tree (AST) of the Solidity code, a symbolic analyzer that detects possible vulnerable states, and a greybox fuzzer that detects vulnerable execution paths. These tools cooperate to generate the results. Finally, MythX provides an analysis report which lists all the weaknesses found in the code, including the exact position of the issue.

Tikhomirov *et al.* [17] provide a comprehensive classification of code issues in Solidity and implement the SmartCheck static analysis tool. The tool translates Solidity source code into an XML-based intermediate representation and checks it against weakness signatures specified by XPath patterns. SmartCheck has its limitations, as detection of some bugs requires more sophisticated techniques such as taint analysis or even manual audit.

Kalra *et al.* [18] present the ZEUS framework to verify the correctness and validate the fairness of smart contracts. They consider correctness as adherence to safe programming practices, while fairness is adherence to an agreed-upon higher-level business logic. ZEUS leverages both abstract interpretation and symbolic model checking, along with the power of constrained horn clauses to quickly verify contracts for safety. A prototype of ZEUS was developed and run against over 22.4K smart contracts. The evaluation indicated that about 94.6% of the contracts (keeping cryptocurrency worth more than 0.5 billion USD) are vulnerable.

Mossberg *et al.* [19] present Manticore, that employs symbolic execution to find unique computation paths in EVM (and ELF) binaries. It checks the traces for vulnerabilities like reentrancy and reachable self-destruct operations, and reports them in the context of the source code.

Mythril [20] is a command-line tool in Python for analyzing smart contracts interactively. It uses symbolic execution, SMT solving and taint analysis to detect a variety of security vulnerabilities. It detects security vulnerabilities in smart contracts built for Ethereum, Hedera, Quorum, Vechain, Roostock, Tron and other EVM-compatible blockchains.

3.2 Online Methods

Azzopardi *et al.* [21] developed the ContractLarva tool for runtime verification of Solidity smart contracts. ContractLarva takes the contract code as well as a contract-related specification as its input, and generates a new contract in the Solidity language that behaves like the original contract and also monitors its runtime behavior against the specification.

Wang *et al.* [22] introduced ContractGuard which is an intrusion prevention system (IPS) for Ethereum

Table 1. Notable Ethereum clients [7]

Client	Language	Operating systems	Networks
Geth [8]	Go	Linux, Windows, MacOS	Mainnet, Sepolia, Görli,
OpenEthereum [9]	Rust	Linux, Windows, MacOS	Mainnet, Sepolia, Görli, and more
NEthermind [10]	C# .NET	Linux, Windows, MacOS	Mainnet, Sepolia, Görli, and more
Hyperledger Besu [11]	Java	Linux, Windows, MacOS	Mainnet, Sepolia, Görli, and more
Trinity [12]	Python	Linux, MacOS	Mainnet, Sepolia, Görli
Erigon [13]	Go	Linux, Windows, MacOS	Mainnet, Sepolia, Görli, and more

smart contracts. The protection is provided in three phases: (1) training, (2) protection, and (3) alarm-verification. ContractGuard's code is embedded in the contracts to profile context-tagged acyclic paths and stop abnormal control flows. It is optimized for a minimal gas consumption with respect to the Ethereum's gas-oriented performance model.

Wang *et al.* [23] introduced the FSFC input-filtering framework for smart contracts. FSFC is designed to allow Ethereum smart contracts to run normally even in the face of attacks. In particular, it dynamically identifies and rejects bad inputs before they are given to contracts for processing. In this approach, a contract owner can add new filters to FSFC and protect his/her contract without disrupting its service. Once a contract is found to be vulnerable, the detection tools can generate the required filters to be added to guard against bad inputs.

Rodler *et al.* [24] introduced Sereum for protecting existing smart contracts againts re-entrancy attacks based on runtime monitoring and validation. It does not require any modification of existing contracts and is able to cover the actual execution flow of a smart contract to detect and prevent re-entrancy attacks.

Torres *et al.* [25] introduced ÆGIS, a dynamic analysis tool that protects smart contracts during runtime. Their idea is to bundle every Ethereum client with a runtime analysis tool, that interacts with the EVM and is capable of interpreting some attack patterns, and reverting transactions that match the patterns. Attack patterns are described using their domain-specific language (DSL), and are voted upon and stored via a smart contract.

Li *et al.* [26] introduced Solythesis, a source to source Solidity compiler which takes a smart contract code and a user specified invariant as the input and produces an instrumented contract that rejects all transactions that violate the invariant. It automatically instruments a Solidity smart contract with custom invariants. Compared to Solidity, its specification language for invariants contains additional features, including quantifiers and sums. It also has a high gas overhead because of the runtime checks.

4 The Proposed Method

An overview of our proposed method is depicted in Figure 1. The contract developer starts the process and analyzes the source code using the existing offline methods, and fixes the identified weaknesses. Then, the contract owner uses our user interface (UI), selects the appropriate guards, and sets their parameters. The UI generates small pieces of code to be added to the contract's source code. Once a transaction is sent to the contract, our enhanced version of the Ethereum client determines the enabled guards, and protects the contract from potential attacks.

The safety guards reassure the contract owner that if there is a vulnerability in the contract, it cannot be exploited. Our method is not bound to specific vulnerabilities or attack methods. Therefore, this method is proactive and preventive and the guards can prevent all related potential attacks even if they exploit zero-day vulnerabilities. In addition, the safety guards have access to all the runtime information which may not be accessible to offline methods and within a contract's code. Our method can be easily applied to private and enterprise implementations of the Ethereum network. Nonetheless, a soft-fork is required for wide-scale use in the Ethereum mainnet.

If our method is not used, the respective restrictions must be implemented in the source code of the contract (if possible). For the following reasons, we believe that the runtime enforcement is a better choice than integrating the guards in the source code:

- The safety checks will be spread out in different parts of the source code. The checks can be easily forgotten or not properly enforced (e.g., in assembly code). In addition, there are possible ways to bypass them in *delegate calls.*
- The checks can complicate the code of the contract and make it less readable.
- There is an about 25KBs limit on the size of the bytecode of a contract. If the checks are

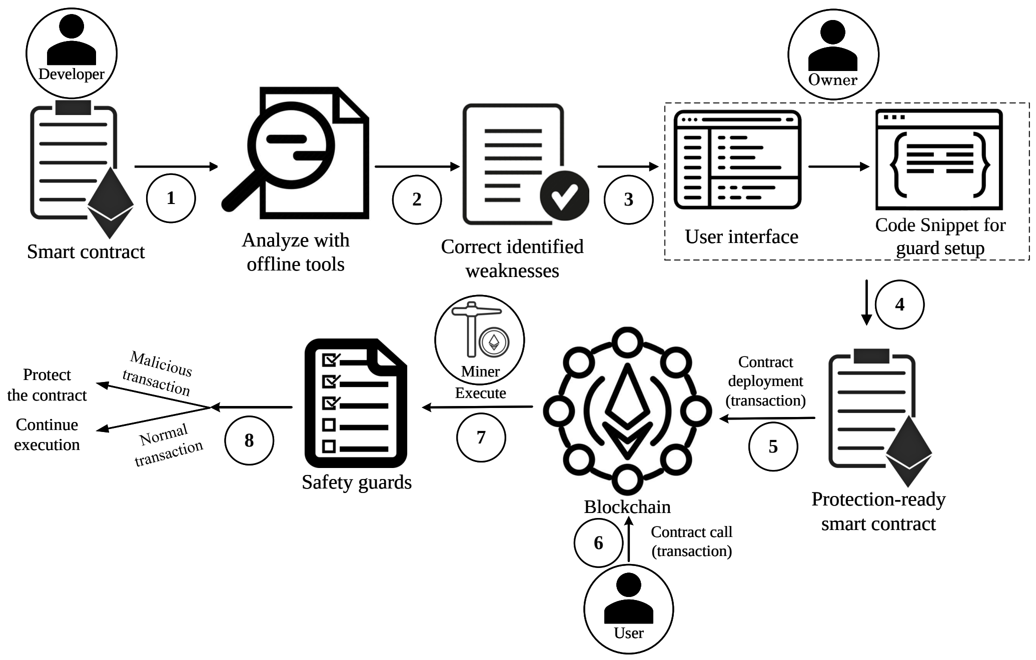

Developer
Smart contract
1
Analyze with offline tools
2
Correct identified weaknesses
3
Owner
User interface
Code Snippet for guard setup
4
Protection-ready smart contract
Contract deployment (transaction)
5
Blockchain
6
Contract call (transaction)
User
Miner
Execute
7
Safety guards
8
Malicious transaction
Protect the contract
Normal transaction
Continue execution

Figure 1. An overview of our proposed method

integrated into the source code, they will make more pressure on the implementation of the main functionality of the contract.

- The checks in the source code will increase gas consumption of all method calls, while in our proposal, the miners get an extra fee (the remaining gas) for the violating transactions.
- Adding the checks to a contract's code in a proper way requires a sufficient level of programming knowledge that the contract owner may not have.
- A wider range of safety guards can be implemented in the runtime than in the source code because the contracts do not have access to all the runtime data.

The last point is an important advantage of our method and needs more explanation. There is data that is not directly accessible by the contract code because it is related to the Ethereum network and its nodes. As an example consider the accurate gas consumption, for which we have also implemented a guard. While it may be possible for a contract to keep an eye on gas consumption in specific points in the contract, but it is not possible to track gas consumption. Other examples are as follows:

- Transactions in the current block and their order
- Network statistics
- All other real-world data is not exposed by EVM to contracts

More detail about our proposal is given subsequently.

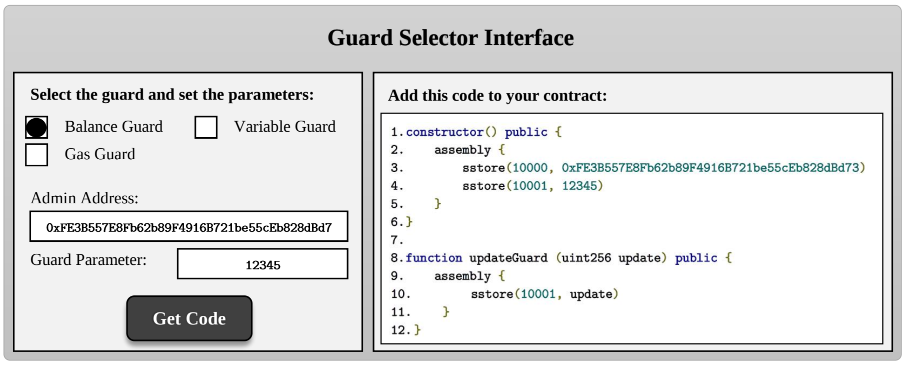
4.1 The User Interface

There is no need for fundamental changes in the contract's source code for using the guards; only the required parameters are necessary to be added to the code. The contract owner selects the guards through a simple user interface (UI) and sets their parameters. Subsequently, small Solidity code snippets are generated to be inserted in the contract's code. Our user interface is similar to Figure 2.

4.2 The Enhanced Runtime

In order to activate and use each guard, it is necessary for the contract owner to insert the guards' parameters in the source code of the contract, so that when the contract is deployed, the parameters are also sent to the network.

The general flow of the guards are depicted in Figure 3. During the processing of a transaction, our enhanced runtime determines if the contract involves some safety guards. If so, their conditions are checked for the transaction and in case of encountering the prohibited conditions, the transaction is aborted and the network state will not be updated. All the remaining gas will also be consumed as a motivation for the miners for implementing the safety guards. Since the guards are implemented at the lowest level and in the runtime, it is not possible to bypass them.


Guard Selector Interface

Select the guard and set the parameters:

● Balance Guard ☐ Variable Guard
☐ Gas Guard

Admin Address: 0xFE3B557E8Fb62b89F4916B721be55cEb828dBd7

Guard Parameter: 12345

Add this code to your contract:

```
constructor() public {
    assembly {
        sstore(10000, 0xFE3B557E8Fb62b89F4916B721be55cEb828dBd73)
        sstore(10001, 12345)
    }
}

function updateGuard (uint256 update) public {
    assembly {
        sstore(10001, update)
    }
}
```

Figure 2. The user interface of the proposed method

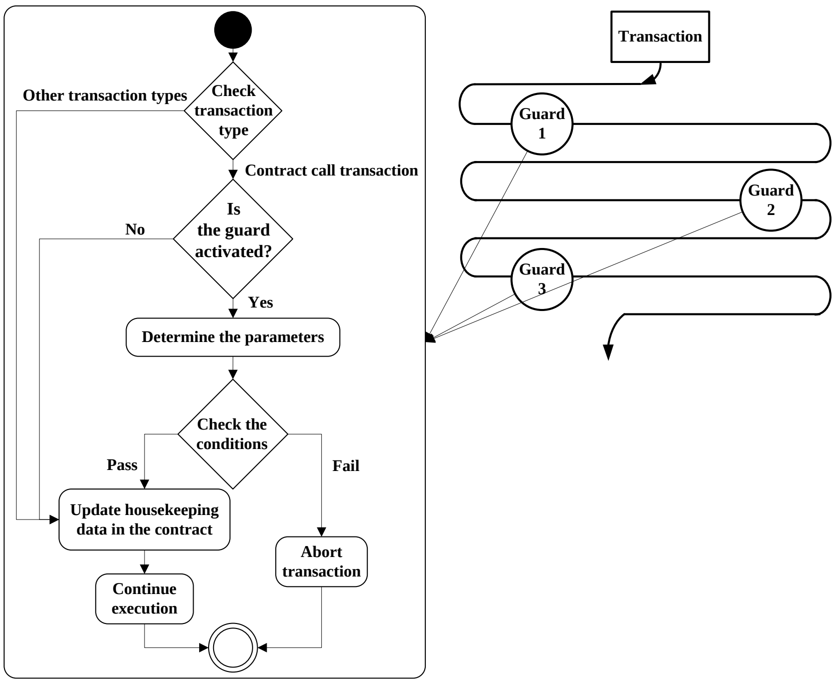


Figure 3. The transaction execution process and the general flow of the guards

4.3 Administration of the Guards

The guard administration feature allows a predefined administrator to update the value(s) of a guard's parameter(s). The feature is optional and can be enabled by defining a guard update function in the contract's code which is generated by the UI. All modifications to the slots containing the parameters of the guards, either by this function or any other way, are checked by the enhanced runtime to be only from the administrator's address (if defined). By the way, manipulation of housekeeping data related to the guards are totally prohibited.

4.4 The Safety Guards

Although the main focus of the paper has been on the idea of using the safety guards, we propose three guards in this section to better showcase our proposal. Nonetheless, the proposal is not limited to these guards, and other safety guards can also be integrated into the implementation. The proposed safety guards are explained in the following subsections. We also state some of the smart contract weaknesses (base on SWC-Registry [27]) that can be protected by each guard.

4.4.1 Contract Balance Guard

The balance guard sets a limit on the amount of withdrawals from the contract during a time interval. If a zero-day vulnerability occurs that can be exploited to completely drain the contract balance, this guard can protect the contract and slow down such an attack. In particular, the guard monitors the amounts decreased from the contract's balance in a predefined time-frame. If the amount exceeds an owner-set threshold, no more transfers are allowed, and the related transactions will be reverted until a new time-frame begins. It gives the contract owner, the opportunity to use compensatory solutions and prevents a fatal attack. Some weaknesses such as SWC-104[2], SWC-105[3] and SWC-107[4] can be protected by this guard.

4.4.2 Contract Variable Guard

This guard protects the storage slots (data fields) of the contract from being changed inadvertently. For example, a variable containing the address of the contract's owner(s) is a typical target for attackers. It is common for smart contract attackers to first exploit a vulnerability to change the owner(s) and then, misuse the authority of the owner(s) for their benefit. This guard can prevent these types of attacks, in the first place. Some weaknesses such as SWC-106[5], SWC-109[6] and SWC-124[7] can be protected by this guard.

4.4.3 Gas Guard

This guard monitors the average gas consumption related to the contract's transactions, and keeps them below an owner-set threshold. This guard would have been used to stop the well-known attack to the FairWin contract which drained the contract's balance. The sign that made researchers suspicious for the attack was unusual and high consumption levels of gas over a period of one month [28]. By using this guard, the attacker's ability for misuse is totally confined. Some weaknesses such as SWC-105[8], SWC-107[9] and SWC-128[10] can be protected by this guard.

4.4.4 Motivation for the Miners

In order to account for the extra processing required for safety guards, we take an approach similar to Wang *et al.* [23]. In particular, we avoid from changing the gas consumption of EVM opcodes but if a transaction is aborted because of violating a guard, the entire gas amount assigned to that transaction is paid to the miners and no gas is refunded to the caller.

By the way, if attackers are able to extract a large amount of asset by exploiting vulnerable contracts, this can adversely affect the cryptocurrency price and subsequently the miners' profits. So, it is good for miners to run guards and try keeping the contracts secure.

4.5 The Guards' Parameters

We reserve specific smart contract storage slots to store the guards' parameters and their housekeeping data. There are a total of 2^{256} free storage slots, and we use at most 7 slots for guards' parameters and 20 slots for housekeeping data (in case all guards are enabled). The guards' parameters are set in the constructor of the contract using the `SStore` EVM assembly instruction.

The balance guard needs one storage slot to store the withdrawals limit amount, the variable guard needs four storage slots to store the sensitive variable's index and three valid values, and the gas guard needs one storage slot to store the gas usage threshold. The gas guard also requires 20 more storage slots to store some housekeeping data. In particular, 10 slots keep block numbers, and 10 others keep the gas usage per block. All the guards share the same parameter which stores the administrator's address (if defined) or a non-zero value (if no administrator defined). Should one of these slots be updated, only the admin is allowed to do so.

The contract owner should ensure that the set parameters reflect the normal behavior of the contract and minimize the chance that a normal transaction is stopped by a guard. This setting can be found during the test phase of the contracts (beta test, before the final release of the contract), and the parameters of the guards should be set accordingly.

4.6 Possible Side Effects

One may be concerned about the side effects of the guards and worry about new attacks which misuse the guards. We elaborate on three possibilities for this:

(1) The contract has not been under attack and normal transactions have been canceled by mistake (false positive). In order to prevent this, the owner should set the guards' parameters, carefully.
(2) An attacker somehow benefits from the feature of canceling suspicious transactions, and arranges conditions so that a certain transaction is not executed successfully. Since it is not possible for the attacker to interfere in the execution of other users' transactions, he/she can only persuade others to execute a transaction that will be canceled. In this case, the guard performs its purpose and properly stops the at-

[2] https://swcregistry.io/docs/SWC-104
[3] https://swcregistry.io/docs/SWC-105
[4] https://swcregistry.io/docs/SWC-107
[5] https://swcregistry.io/docs/SWC-106
[6] https://swcregistry.io/docs/SWC-109
[7] https://swcregistry.io/docs/SWC-124
[8] https://swcregistry.io/docs/SWC-105
[9] https://swcregistry.io/docs/SWC-107
[10] https://swcregistry.io/docs/SWC-128

tacker.

(3) If the performance of the transaction execution is dramatically affected because of the guards, they may limit the blockchain's TPS (transactions per second). We will later demonstrate that our guards have a little overhead. However, in the case that some guards other than the proposed ones are implemented in the runtime, great care should be taken about their implementation efficiency. In addition, the guards should be enabled only when necessary, and the other offline code analysis methods should be used as much as possible to prevent vulnerabilities.

We explain the second point above as follows. Consider a case where the attacker aims to prevent the execution of some transactions by misusing the *balance guard*. In this case, in the presence of a balance guard, the attacker can make a weaker attack than normal, because:

- The guards prevent the rapid discharge of the contract balance, so when there is a vulnerability in the contract, it gives the contract owner the opportunity to use compensatory solutions and prevents a fatal attack.
- The attacker's offensive action has a temporary effect on the contract and by passing the time frame, the users can withdraw from the contract again.
- There is no guarantee that the attacker can repeat his attack again at the beginning of the next frame, because the order of transactions in a block and what transaction goes in a block is not set by the attacker.
- This attack has costs for the attacker because he must first deposit a balance in the contract to be able to withdraw that.

5 Implementation

Performance and evaulation of safety guards depends on each specific guard and its method of implementation. It is also essential to ensure that the implementation is done with minimum overhead. The guards' parameters themselves usually do not take up much space, but can be troublesome if a guard needs to lot of storage slots to maintain state. In the following subsections, we explain the details of our implementation of the three proposed guards in the open-source Hyperledger Besu client [11]. Our main focus has been on the idea of using the safety guards and introduced these three sample guards for a better understanding of the idea and evaluations. More guards should be implemented according to our proposal to make a full-featured product.

5.1 Reserved Slots

In our implementation, the reserved slots for the guards are fixed for all contracts, and each slot is used to store the value of a specific parameter. For example, if we consider `S1` to `S7` as the reserved slots, `S1` is always used to store the address of the guards' admin. We also look at the guard admin slot to determine whether the contract owner has enabled any guards or not. In particular, the storage slots are empty by default, and a non-zero value in one of the reserved ones indicates that the related guard is enabled.

5.2 Implementation of the Balance Guard

We limit the time-frame for the balance guard according to the number of confirmed blocks. In this implementation, we keep track of the total amount of withdrawals from the contract's balance in successive block groups each containing a hundred block. The guard is implemented in the `MainnetTransactionProcessor` class of the Besu client, and a total of three storage slots are reserved for the guard—one for the amount limit, and two additional slots to store the last block number and the accumulative amounts of withdrawals in the current time-frame. If a transaction involves a transfer, its block number is divided by 100 and compared to the last transfer block number (also divided by 100). If the results are different, it means that the new transaction is in a new frame and the withdrawal limit have to be reset (e.g., blocks 199 and 200 are in different time-frames). Otherwise, the frame has not ended yet, and the transfer is validated against the set threshold. If the threshold is exceeded, the transaction will be aborted and the network state will not be updated. All the remaining gas will also be consumed.

5.3 Implementation of the Variable Guard

Whenever a value is assigned to a contract variable (slot), or its value is updated, the `SStore` opcode is executed in the EVM. Therefore, we implemented the variable guard inside the `SStoreOperation` class of Besu (besides other EVM operations). The inputs to `SStore` are a slot number (key) and a new value. The implementation simply checks the inputs against the guard parameters. If the slot number is different or the new value is valid, the normal execution will proceed. Otherwise, the transaction will be aborted and the network state will not be updated. All the remaining gas will also be consumed. In our implementation, the contract owner can specify at most three different valid values for a variable. Hence, the guard requires four parameters in addition to the admin address. The first parameter defines the storage slot of the

Table 2. An example "block number – gas usage" array

Block Number	Gas Usage
110	63,900
101	21,300
...	...
129	42,600

sensitive variable, and the other parameters specify the three valid values. Nonetheless, the guard can also be implemented for range checks in a similar way.

5.4 Implementation of the Gas Guard

We implemented the gas guard in the `MainnetTransactionProcessor` class of the Besu client. In this implementation, the gas consumed in the last 10 blocks (with transactions targeting the protected smart contracts) is always examined (10 blocks are considered as a simple implementation and more blocks can be considered). The contract owner sets the desired limit on average gas consumption in the guard's parameter. If several transactions are sent to the contract in one block, the sum of their consumed gas is considered as the gas consumption of the contract in the block. In the case that no transaction is sent to the contract in a block, a zero gas value will be accounted for that block.

The block numbers and the gas consumed in each block are stored in the contract storage slots, and the guard has access to this array of data (see Table 2). In this array, the remainder of dividing the block number by 10 determines the row to be updated or overwritten. If the existing data is older than the 10 preceding blocks, it will be overwritten. Otherwise, the value will be updated. The gas guard uses this data to calculate the average gas consumption in the last 10 blocks and compare it with the threshold value. If the amount exceeds the threshold, the transaction will be cancelled, and all the remaining gas is consumed. Otherwise, the gas usage and the block number of the transaction are stored in the contract storage for future checks.

6 Case Study

This section provides practical examples of using each guard and shows how a guard can protect a contract.

6.1 Contract Balance Guard Case Study

This guard limits the Ether withdrawal from the contract's account. We consider an initial contract which is listed in Contract 1. This contract has two functions. The `pay` function is defined to deposit to the contract's account and the `withdraw` function is defined for withdrawal. By enabling the contract balance guard, it will monitor the changes in the contract's balance during a 100-block time-frame.

```
pragma solidity ^0.4.0;
contract GUARD1 {
  function pay() public payable {
  }
  function withdraw(uint32 amount) public {
    msg.sender.transfer(amount);
  }
}
```

Contract 1. The initial contract before enabling the balance guard

For using the balance guard, the owner enters the desired threshold in the unit of Wei (10^{-18} Ether) and the address of the guard admin in the UI. In our example, the address of the guard admin is `0xFE3B...Bd73` and the amount of the threshold is 5×10^9 Wei. After entering these values, the user interface generates the required Solidity code to be added to the contract's source code. The final source code is listed in Contract 2. The guard parameters are set in the constructor of the contract (EVM assembly), and a function named `updateGuard1` has been added to the contract. The admin can update the withdrawal threshold by calling this function. The calls to the method from other addresses will be aborted.

```
pragma solidity ^0.4.0;
contract GUARD1 {
  constructor() public {
    assembly {
          sstore(10000,
 0xFE3B557E8Fb62b89F4916B721be55cEb828dBd73)
          sstore(10001, 5000000000) }
  }
  function pay() public payable {
  }
  function withdraw(uint32 amount) public {
    msg.sender.transfer(amount);
  }
  function updateGuard1(uint256 update) public {
    assembly {
          sstore(10001, update)
        }
  }
}
```

Contract 2. The contract's code with the balance guard

6.2 Contract Variable Guard Case Study

This guard controls the changes in the sensitive storage variables of the smart contract. We consider an initial contract which is listed in Contract 3. In this contract, the contract's owner address is stored in a variable called `owner`. There is also a function named `changeOwner` that can be called to change the address of the contract owner.

```
pragma solidity ^0.4.0;
contract GUARD2 {
  address owner;
  constructor() public {
    owner = msg.sender;
  }
  function changeOwner() public {
    owner = msg.sender;
  }
}
```

Contract 3. The initial contract before enabling the variable guard

For demonstration purposes, we consider that only the following three addresses are allowed to be set as the `owner`: `0xFE3B...Bd73`, `0xF17F...B732`, and `0xAC8A...BE17`. In order to enable the variable guard, the owner enters the name of the sensitive variable, the allowed values, and the address of the guard admin (`0xFE3B...Bd73`) in the UI. Subsequently, it generates a few lines of Solidity code to be added to the contract's code.

The final source code is listed in Contract 4. The guard parameters are placed in the constructor of the contract (EVM assembly), and a function named `updateGuard2` has been added to the contract. Only the admin can update the three allowed values by calling the update function.

6.3 Gas Guard Case Study

This guard cancels a transaction when the average gas consumption exceeds an owner-defined threshold. As an example, we consider an initial contract which is listed in Contract 5. For using the gas guard, the contract owner specifies the threshold amount. For instance, he/she may choose 200,000 gas, and enter it in the UI along with the admin address (`0xFE3B...Bd73`). The UI generates the required Solidity code, and the owner integrates them into the contract's code. The threshold amount and the admin address are stored in their respective slots in the constructor, and the `updateGuard3` function is added to the contract for updating the threshold amount (only by the admin). The resulting code is listed in Contract 6.

```
pragma solidity ^0.4.0;
contract GUARD2 {
  address owner;
  constructor() public {
    owner = msg.sender;
       assembly {
      sstore(10000,
 0xFE3B557E8Fb62b89F4916B721be55cEb828dBd73)
      sstore(10002, owner_slot)
      sstore(10003,
 0xFE3B557E8Fb62b89F4916B721be55cEb828dBd73)
      sstore(10004,
 0xF17F52151EBEF6C7334FAD080C5704D77216B732)
      sstore(10005,
 0xAC8A541C8FC62b89F7956B141be75cED828ABE17) }
  }
  function changeOwner() public {
    owner = msg.sender;
  }
  function updateGuard2(uint256 update1,
   uint256 update2, uint256 update3) public {
    assembly {
      sstore(10003, update1)
      sstore(10004, update2)
      sstore(10005, update3) }
  }
}
```

Contract 4. The contract's code with the variable guard

```
pragma solidity ^0.4.0;
contract GUARD3 {
  uint256 a = 1;
  uint256 b = 2;
  function fun1() public view returns (uint256) {
    return a;
  }
  function fun2() public view returns (uint256) {
    return b;
  }
}
```

Contract 5. The initial contract before applying the gas guard

6.4 The SimpleDAO Case

This section introduces a real-world vulnerable contract and uses a safety guard for its protection. This contract, the `SimpleDAO`[11], is a simplified version of `TheDAO` contract which raised about 150M USD before being attacked in 2016. `SimpleDAO` allows participants to donate Ether to fund other contracts at their choice, and the selected contracts can later withdraw the allocated funds. The vulnerable code [29] is listed in Contract 7.

The re-entrancy vulnerability of this contract allows an adversary to steal all its Ether. The first

[11] https://swcregistry.io/docs/SWC-107#simple_daosol

```
pragma solidity ^0.4.0;                                1
contract GUARD3 {                                      2
  uint256 a = 1;                                       3
  uint256 b = 2;                                       4
  constructor() public {                               5
    assembly {                                         6
          sstore(10000,                                7
 0xFE3B557E8Fb62b89F4916B721be55cEb828dBd73)           8
          sstore(10007, 200000) }                      9
  }                                                    10
  function fun1() public view returns (uint256) {      11
    return a;                                          12
  }                                                    13
  function fun2() public view returns (uint256) {      14
    return b;                                          15
  }                                                    16
  function updateGuard3(uint256 update) public {        17
    assembly {                                         18
          sstore(10007, update) }                      19
  }                                                    20
}                                                      21
```

Contract 6. The contract's code with the gas guard

```
pragma solidity 0.4.24;                                1
contract SimpleDAO {                                   2
  mapping (address => uint) public credit;             3
  function donate(address to) payable public{          4
    credit[to] += msg.value;                           5
  }                                                    6
  function withdraw(uint amount) public{               7
    if (credit[msg.sender]>= amount) {                 8
      require(msg.sender.call.value(amount)());        9
      credit[msg.sender]-=amount;                      10
    }                                                  11
  }                                                    12
  function queryCredit(address to) view public         13
        returns(uint) {                                14
    return credit[to];                                 15
  }                                                    16
}                                                      17
```

Contract 7. The `SimpleDAO` contract before applying the gas guard

step of the attack is to publish another contract, the `Mallory` contract, which is listed in Contract 8. Then, the adversary donates some Ether for `Mallory`, and invokes the `Mallory`'s `fallback`function. Subsequently, the function invokes `withdraw`, and the Ether is transferred to `Mallory`. Now, the function call used for this purpose has the side effect of invoking the `Mallory`'s `fallback` again, which maliciously calls back `withdraw` for a second time. Note that the `withdraw` function transfers the fund before updating the `credit`, and the check at line 11 succeeds again [29]. The attack continues in the same fashion until the attacker-set limit on the number of re-entrant calls is reached.

```
pragma solidity 0.4.24;                                1
contract Mallory {                                     2
  SimpleDAO public dao = SimpleDAO(0x354...);          3
  address owner;                                       4
  uint n;                                              5
  constructor() public {                               6
    owner = msg.sender;                                7
  }                                                    8
  function () {                                        9
    if (n > 0) {                                       10
        n--;                                           11
        dao.withdraw(dao.queryCredit(this));           12
    }                                                  13
  }                                                    14
  function attack(uint limit) {                        15
    n = limit-1;                                       16
    dao.withdraw(dao.queryCredit(this));               17
  }                                                    18
  function getJackpot (){                              19
    owner.send(this.balance);                          20
  }                                                    21
}                                                      22
```

Contract 8. The `Mallory` contract

The gas guard is a good choice for protecting the `SimpleDAO` contract and limiting the attacker's benefit. In order to use the guard, we set a limit of 1M gas for the guard. The modified source code is listed in Contract 9.

When the attacker wants to exploit the (existing) re-entrancy vulnerability, after a few transactions, the gas consumption limit will be exceeded and the attacker's transactions will be failed. In the meantime, the contract stakeholders have some time to ban any further deposits to the contract.

7 Evaluation

In this section, we demonstrate the results of our evaluations and compare our proposed method with some other smart contract protection methods. First, a qualitative comparison is made between the features of the proposed method and the other methods, then the details of the costs and overheads are provided.

7.1 Comparison with the Related Works

In order to have a fair comparison, we selected the most important criteria and discuss them one-by-one in the following. The evaluations related to the overheads are performed in a separate subsection, and we do not discuss them here. Table 3 summarizes the results of the comparison.

Table 3. Comparison between the proposed method and the other methods

Tool		Protection Time	Changes in the Contract	Changes in the Runtime	Overhead	Customizability	Practical Restrictions	User Friendliness
Proposed Mechanism		After Deployment	Little Changes	Little Changes	Low	Yes	No	Yes
Offline Tools [14]-[20]		Before Deployment	Little to Extensive Changes	No Change	No Overhead	No	Yes	No
Online Tools	**Contract Larva** [21]	After Deployment	Extensive Changes	No Change	Medium	Yes	Yes	No
	Contract Guard [22]	After Deployment	Extensive Changes	Little Changes	High	No	Yes	No
	FSFC [23]	After Deployment	Little Changes	Extensive Changes	Medium	No	No	No
	Sereum [24]	After Deployment	No Change	Extensive Changes	Low	No	Yes	Yes
	ÆGIS [25]	After Deployment	No Change	Extensive Changes	*N/A*	Yes	No	No
	Solythesis [26]	After Deployment	Extensive Changes	No Change	High	Yes	Yes	No

```
pragma solidity 0.4.24;
contract SimpleDAO {
  mapping (address => uint) public credit;
  constructor() public {
    creator = msg.sender;
    assembly {
          sstore(10000,
 0xFE3B557E8Fb62b89F4916B721be55cEb828dBd73)
          sstore(10007, 1000000) }
  }
  function donate(address to) payable public{
    credit[to] += msg.value;
  }
  function withdraw(uint amount) public{
    if (credit[msg.sender]>= amount) {
      require(msg.sender.call.value(amount)());
      credit[msg.sender]-=amount;
    }
  }
  function queryCredit(address to)
    view public returns(uint){
    return credit[to];
  }
  function updateGuard3(uint256 update) public {
    assembly { sstore(10007, update) }
  }
}
```

Contract 9. The SimpleDAO contract with the 1M gas guard

7.1.1 Protection Time

Our proposal is an online method, and starts working right after the contract is deployed in the network. This makes it possible to access real runtime information, and protect the contract against real vulnerabilities, whereas offline methods cannot access such information which may be critical in spotting the misuses.

7.1.2 Changes in the Contract

In our proposed method, some parameters (in the form of short Solidity code snippets) are added to the contract's source code. However, the changes are not fundamental which is not the case in all the methods that change the contract's code. We keep the structure of the final contract the same as the original contract and do not hinder its readability. This positive aspect also opens the possibility for the UI to generate the code snippets for other smart contract programming languages.

7.1.3 Changes in the Runtime

In our method, a protection code is added to the runtime. It is more effective in protecting the contract than the offline tools and methods that add a protection code to the contract's source code. By the way, our changes to the runtime do not conflict with the original Ethereum runtime, and can be adopted by the Ethereum public network in the form of a soft fork. As we mentioned before, the miners receive all the remaining gas for transactions aborted by our guards, it can be considered as a motivation for running our improved client. In fact, it is not necessary to use a guard for all contracts, and they can be used in normal conditions without using the guards.

7.1.4 Customizability

In our proposal, all contracts are not protected in the same way. The owners of the contracts can select the guards to be used (or do not enable any guards at all), and adjust their parameters according to the nature of their contracts. Therefore, the conditions are checked differently for each contract. Also, this method can protect contracts against different attack types, but some tools such as Sereum [24] only focus on a single vulnerability.

7.1.5 Practical Restrictions

Due to the small additions to the contract's code, our proposed method can also be used for big contracts. In other words, we do not decrease the maximum possible size of code by our protection method. In contrast, some other (online) protection methods are not applicable to big contracts due to the large amounts of code they require to be added to the source code.

7.1.6 User Friendliness

Most methods require deep knowledge of smart contract programming, but in our method there is no need for such a deep knowledge, and our user interface generates the required code, and they can simply be added to the contract's code with a minimum programming knowledge.

7.2 Performance Evaluation

We evaluate the performance of our implementation with respect to (1) the increased code size for incorporating the guard parameters (deployment gas cost), and (2) the extra processing time for guards (execution overhead for the miners). In order to perform the evaluations, we ran a local test Ethereum network consisting of three Besu clients and deployed our test contracts from the preceding section on the network. Then, we called several methods of the deployed contracts, and measured the gas consumption and transaction processing time of the transactions. We followed this process for an original Besu client as well as our improved version, and compared the results under the same conditions. In the following, we report and discuss the results.

7.2.1 Transaction Processing Time

In this section, we report and examine the processing time of different transactions and compare them with the original Besu client without implementing the safety guards. Our proposal should have little impact on normal users. Therefore, we evaluated the processing time for three cases: (1) sending a contract deployment transaction to the network; (2) sending a transaction to a contract without any guards; and (3) sending a transaction to a contract with a guard that do not violate any of the terms of the guard. For this purpose, we launched a private network with three nodes running the original version of Besu, and another version of the same network by our improved version (denoted by Besu+). For each guard, we sent a transaction for 30 times under the same conditions to both the normal and the improved networks, and measured the average transaction processing time.

Table 4. The time overhead for method calls to contracts with the guards in the upgraded environment

Guard / Node	Balance	Variable	Gas
Node 1	5.65%	24.82%	69.23%
Node 2	7.03%	23.72%	70.18%
Node 3	4.06%	29.82%	60.99%

The experiments were done on a 2.5GHz Intel 4710HQ processor with 4-cores and 8GBs of RAM running the Ubuntu 20.04 OS. The version of the original Hyperledger Besu client was 20.10.2. Figure 4 demonstrates the method call processing time (without any guard), and Figure 5 illustrates the contract deployment processing time for the different guards for each of the nodes in the two environments.

These results demonstrate that in the two cases of (1) sending a transaction to a contract without a guard and (2) deploying a new contract, the two environments have a processing time difference of less than 10%. It is noteworthy that according to our observations, this difference is quite negligible and fits in the error range of the measurements. We also measured the method call processing time for the transactions which were sent to the contracts with the guards. The results are depicted in Figure 6 and the overhead is reported in Table 4. The latter represent the average processing time of one transaction. According to Table 4, the added overhead when using the balance guard is about 5% and for the variable guard is about 30%. However, the overhead is about 70% for the gas guard because it performs more operations to store and calculate the gas consumption for each block.

We have changed very limited parts of the runtime code, and as much as possible, we have tried to optimize the changes to reduce the overhead of the newly added processes. In particular, the overhead of execution of a guard in comparison to the original Besu code is not dependent to guard parameters or user input. In all guards only a few values are stored in storage slots and a small set of conditions are checked for each transaction to check for signs of an attack. It should be noted that in the runtime, many conditions are checked to validate each transaction by default, and we have added only a limited number of new conditions to these cases. Also, the necessary tests have been done on our codes to identify and fix their possible bugs and make them as bug-free and vulnerability-free as possible.

Our evaluations were measured for very simple and basic contracts (with a size between 500 and

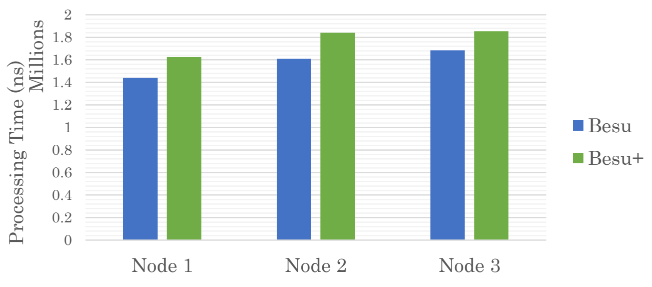
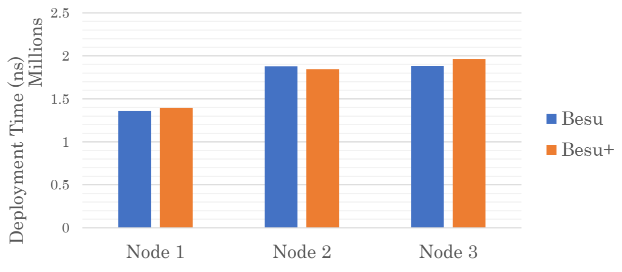
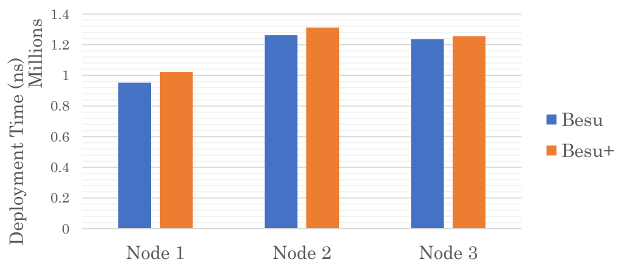


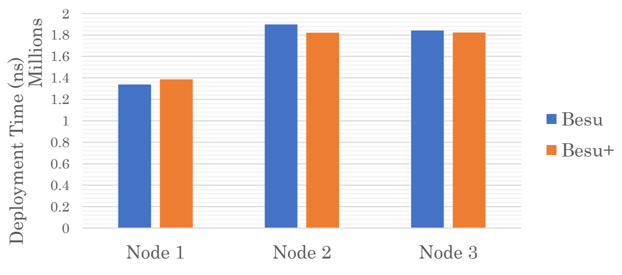
Figure 4. Method call processing time for contracts without any guards in normal and upgraded environments



(a) Deployment time for the balance guard

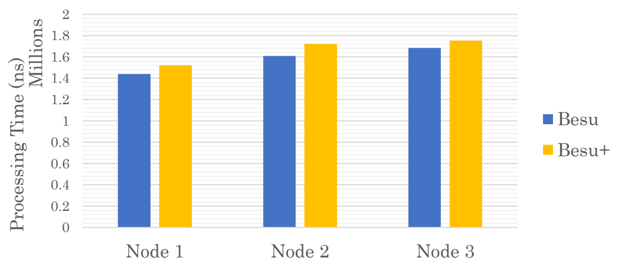


(b) Deployment time for the variable guard

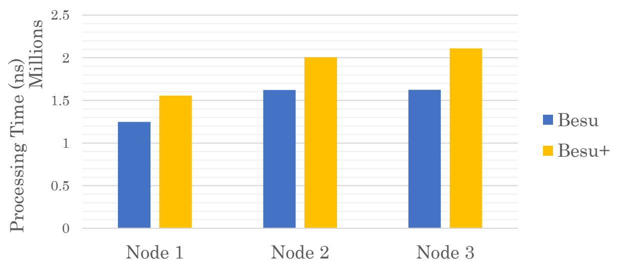


(c) Deployment time for the gas guard

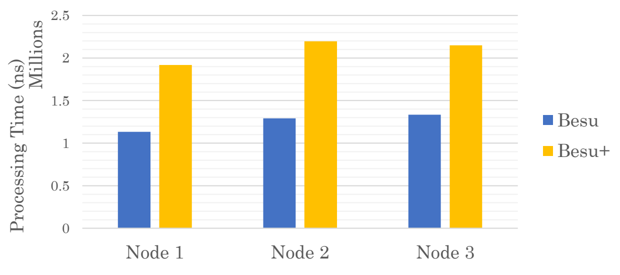
Figure 5. Contract deployment processing time in normal and upgraded environments



(a) Transaction processing time for the balance guard



(b) Transaction processing time for the variable guard



(c) Transaction processing time for the gas guard

Figure 6. Method call processing time for contracts with a guard in normal and upgraded environments

600 bytes). Thus, the percentage of the overhead is more than the case with complex and larger contracts which generally have higher processing times. To better highlight this, we repeated our measurements for the `SimpleDAO` contract with the gas guard (see Contract 9). The results are depicted in Figure 7 and demonstrate that the overhead in this case was greatly reduced to about 30%.

In order to compensate for the extra processing overhead of our guards for the miners, we took an approach similar to Wang *et al.* [23] and did not modify the existing EVM gas costs. Instead, when a guard aborts a possible misuse or attack, the miner will consume all the remaining gas even if the transaction is half-way aborted.

7.2.2 Gas Consumption

The transaction processing fees in the Ethereum are paid by gas, and deducted from the account of the initiator of the transaction. In particular, contract owners pay an extra deployment cost (gas) for incorporating our code in their contracts (initializing the guards' parameters and adding the guard administration functions).

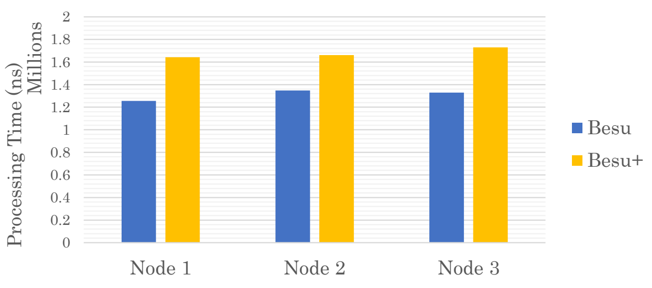


Figure 7. Method call processing time for the `SimpleDAO` contract with the gas guard (see Contract 9) in normal and upgraded environments

In order to evaluate the amount of the added gas, we deployed several contracts without any guards and then, the same transactions were sent with the guard parameters. The results showed that for sending the parameters of the balance, variable, and gas guards, an extra amount of gas equivalent to 71,400, 141,100, 71,400 are added to the deployment transaction, respectively. However, due to the fact that the value of the parameter entered by the owner also affects the amount of gas required to deploy the contracts, the above values may increase or decrease slightly depending on the different values of the parameters. This overhead exists in all other methods that protect contracts during runtime.

Table 5 summarizes the results of the overheads comparison. It should be noted that the mentioned overhead is a fixed value for some tools, and the average or even the worst for the others. We have measured the overhead of our proposed method in the worst case (the maximum overhead) and reported the results in Table 3. Our overhead can be considered as low, but the overheads of the other methods are considered as medium and high.

There is another situation in which the gas costs are different in our improved runtime. In particular, if a transaction is half-way aborted because of a violation in any of the guards, the entire amount of gas allocated to that transaction will be paid to the miners, and nothing is refunded to the caller. Nonetheless, both the users and the owner benefit from applying our safety guards. In particular, the owner makes his/her contract more secure, and the users are ensured that their assets are not misused.

8 Conclusions

Today, with the fast growth of Ethereum smart contracts, their security has become an important issue. In this regard, we introduced a method to protect contracts during runtime by introducing and using some safety guards which are implemented in the Ethereum miners. Using our method, the contract owner specifies a number of guards for the contract and sets their parameters inside the contract code.

Table 5. Overheads of our proposed method and other online methods.

Overhead / Tool	Transaction Gas Cost	Deployment Gas Cost	Execution Time
Proposed Mechanism	No Overhead	<10%	Worst: G1: 7.03% G2: 29.82% G3: 70.18%
Contract Larva [21]	> 20.91% < 225.39%	*N/A*	*N/A*
Contract Guard [22]	Average: 28.27% Worst: 86.0%	Average: 36.14% Worst: 61.23%	*N/A*
FSFC [23]	Average: 1.03 %	*N/A*	Linux: 29.81% Windows: 32.49%
Sereum [24]	*N/A*	*N/A*	Average: 9.6%
ÆGIS [25]	*N/A*	*N/A*	*N/A*
Solythesis [26]	Average: 77.8 %	*N/A*	*N/A*

Our method does not need deep programming knowledge and has low overhead.

In the future, new guards can be developed to prevent more attacks, specially guards that focus on data that is not accessible through the contract source code to better show the point of difference of this method. In addition, our method can be investigated to be applied to smart contract platforms other than the Ethereum. However, not all guards are applicable to other smart contract platforms, mainly because of the differences between the semantics of smart contracts and their underlying operation. Although restricting normal users is one of the inherent limitations of preventive methods, it should be noted that the safety guards need to be used carefully so as not to restrict normal users as much as possible. The guards only work for contracts with ownership semantics, which is true for many, but not all smart contracts. Also in our implementation, the protection mechanisms behind the balance and gas guards are modeled after anomaly detection and it might be possible to avoid them using progressively increasing attacks. However, with the guards being activated, the attacker will be able to perform a weaker attack.

References

[1] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf, December 2008. [Online; accessed 07-October-2021].

[2] Nick Szabo. Smart Contracts: Building Blocks for Digital Markets. https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html, 1996. [Online; accessed 07-October-2021].

[3] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper, 2014. [Online; accessed 07-October-2021].

[4] Solidity Programming Language. https://soliditylang.org. [Online; accessed 07-October-2021].

[5] Ethereum Smart Contracts Anatomy. https://ethereum.org/en/developers/docs/smart-contracts/anatomy/. [Online; accessed 07-October-2021].

[6] Ethereum Transactions. https://ethereum.org/en/developers/docs/transactions. [Online; accessed 07-October-2021].

[7] Ethereum Nodes and Clients. https://ethereum.org/en/developers/docs/nodes-and-clients. [Online; accessed 07-October-2021].

[8] Geth, Official Golang implementation of the Ethereum protocol. https://github.com/ethereum/go-ethereum. [Online; accessed 07-October-2021].

[9] OpenEthereum, Fast and feature-rich multi-network Ethereum client. https://github.com/openethereum/openethereum. [Online; accessed 07-October-2021].

[10] Nethermind, .NET Core Ethereum client. https://github.com/NethermindEth/nethermind. [Online; accessed 07-October-2021].

[11] Hyperledger Besu, An open-source Ethereum client. https://github.com/hyperledger/besu. [Online; accessed 07-October-2021].

[12] The Trinity Ethereum Client. https://github.com/ethereum/trinity. [Online; accessed 07-October-2021].

[13] The Erigon Ethereum Client. https://github.com/ledgerwatch/erigon. [Online; accessed 07-October-2021].

[14] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *The 2016 ACM SIGSAC Conference*, pages 254–269, 10 2016.

[15] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *The 2018 ACM SIGSAC Conference*, pages 67–82, 10 2018.

[16] MythX, Smart contract security tool for Ethereum. https://mythx.io/. [Online; accessed 07-October-2021].

[17] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16, 05 2018.

[18] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. Zeus: Analyzing safety of smart contracts. In *Network and Distributed System Security Symposium*, 01 2018.

[19] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189, 2019.

[20] Bernhard Mueller. Smashing ethereum smart contracts for fun and real profit. In *9th HITB Security Conference*, 2018.

[21] Gordon Pace, Joshua Ellul, and Shaun Azzopardi. Monitoring smart contracts: Contractlarva and open challenges beyond. In *The 18th International Conference on Runtime Verification*, 11 2018.

[22] Xinming Wang, Jiahao He, Zhijian Xie, Gansen Zhao, and Shing-Chi Cheung. Contractguard: Defend ethereum smart contracts with embedded intrusion detection. *IEEE Transactions on Services Computing*, PP:1–1, 10 2019.

[23] Zeli Wang, Weiqi Dai, Kim-Kwang Raymond Choo, Hai Jin, and Deqing Zou. Fsfc: An input filter-based secure framework for smart contract. *Journal of Network and Computer Applications*, 154:102530, 2020.

[24] Michael Rodler, Wenting Li, Ghassan Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. In *Proceedings 2019 Network and Distributed System Security Symposium*, 01 2019.

[25] Christof Torres, Mathis Baden, Robert Norvill, and Hugo Jonker. ÆGIS: Smart shielding of smart contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2589–2591, 11 2019.

[26] Ao Li, Jemin Choi, and Fan Long. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 438–453, 06 2020.

[27] SWC Registry, Smart Contract Weakness Classification and Test Cases. `https://swcregistry.io/`. [Online; accessed 07-October-2021].

[28] The Collapse of FairWin's $125m Ponzi Scheme. `https://medium.com/@PhABC/the-collapse-of-fairwins-125m-ponzi-scheme-61a66b273420`. [Online; accessed 07-October-2021].

[29] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on Ethereum smart contracts. In *International Conference on Principles of Security and Trust*, pages 164–186, 03 2017.

Morteza Amirmohseni is a member of the Blockchain Laboratory at Electrical and Computer Engineering Faculty of Tarbiat Modares University. He holds an M.S. degree in Computer Engineering from Tarbiat Modares University, and a bachelor degree in Computer Engineering from Tehran University. His research interests include Blockchain and Smart Contracts Security, Web and Mobile Applications Security, and API Security in general.

Sadegh Dorri Nogoorani is an assistant professor and the head of the Blockchain Laboratory at Electrical and Computer Engineering Faculty of Tarbiat Modares University. He holds an M.S. in Computer Networks, and a Ph.D. in Computer Engineering from Sharif University of Technology. His research interests are Blockchain and Distributed Ledger technology, Security, Privacy, Trust, and Distributed Systems Design in general.