An Adaptive Gas Cost Mechanism for Ethereum to Defend Against Under-Priced DoS Attacks

Ting Chen[1,2], Xiaoqi Li[2], Ying Wang[1], Jiachi Chen[2], Zihao Li[1], Xiapu Luo[2(✉)], Man Ho Au[2], and Xiaosong Zhang[1]

[1] Cyber Security Research Center, University of Electronic Science and Technology of China, Chengdu 611731, China
{brokendragon,johnsonzxs}@uestc.edu.cn, 769836805@qq.com, gforiq@qq.com

[2] Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong
lee1843@gmail.com, chenjiachi317@gmail.com, {csxluo,csallen}@comp.polyu.edu.hk

Abstract. The gas mechanism in Ethereum charges the execution of every operation to ensure that smart contracts running in EVM (Ethereum Virtual Machine) will be eventually terminated. Failing to properly set the gas costs of EVM operations allows attackers to launch DoS attacks on Ethereum. Although Ethereum recently adjusted the gas costs of EVM operations to defend against known DoS attacks, it remains unknown whether the new setting is proper and how to configure it to defend against unknown DoS attacks. In this paper, we make the *first* step to address this challenging issue by first proposing an emulation-based framework to automatically measure the resource consumptions of EVM operations. The results reveal that Ethereum's new setting is still not proper. Moreover, we obtain an insight that there may always exist exploitable under-priced operations if the cost is fixed. Hence, we propose a novel gas cost mechanism, which dynamically adjusts the costs of EVM operations according to the number of executions, to thwart DoS attacks. This method punishes the operations that are executed much more frequently than before and lead to high gas costs. To make our solution flexible and secure and avoid frequent update of Ethereum client, we design a special smart contract that collaborates with the updated EVM for dynamic parameter adjustment. Experimental results demonstrate that our method can effectively thwart both known and unknown DoS attacks with flexible parameter settings. Moreover, our method only introduces negligible additional gas consumption for benign users.

1 Introduction

Being the second largest blockchain [8], Ethereum distinguishes itself by its Turing-complete execution environment (i.e., EVM) [19] that can run various applications through smart contracts. Besides transferring money, transactions

J. K. Liu and P. Samarati (Eds.): ISPEC 2017, LNCS 10701, pp. 3–24, 2017.
https://doi.org/10.1007/978-3-319-72359-4_1

in Ethereum are also involved in deploying and invoking smart contracts. To ensure that the execution of smart contracts will be terminated eventually, Ethereum charges *gas* (i.e., execution fee) from transaction senders, and lets it be part of the rewards to miners for executing smart contracts. In particular, gas serves as a protection mechanism against resources abusing in case executing certain smart contracts consumes lots of computing resources. The money paid for executing an EVM operation (e.g., addition, multiplication, reading the balance of an account) is the multiplication of the *gas price* with the *gas cost* of that operation, where the gas price indicates the value of one unit of gas and the gas cost of an EVM operation stands for the units of gas required to execute the operation. The gas cost is determined by the EVM in Ethereum client, and the gas price can be set by transaction senders. Every transaction has a *gas limit*, dubbed TGL (Transaction Gas Limit), so that the execution of a smart contract will trigger an *out-of-gas* exception if the execution requires more gas than the TGL. Ethereum attempts to associate EVM operations' gas costs proportionally to the computing resources needed to execute them [19], because a proper setting of gas costs can give miners proper awards and thwart DoS attackers who aim at wasting a large amount of resources.

However, it is non-trivial to properly set the gas cost of each operation because it requires a deep understanding of EVM internals, an accurate measurement of resource consumptions by EVM operations, and the awareness of the market price for different types of computing resources (e.g., CPU, memory, etc.). Failing to select suitable gas costs for EVM operations gives attackers opportunities to launch DoS attacks on Ethereum at low cost by exploiting under-priced operations. An operation is regarded as under-priced if its gas cost is lower than what it should be. Actually, two DoS attacks exploiting such operations were discovered in 2016, which repeatedly execute two under-priced operations, namely `EXTCODESIZE` [7] and `SUICIDE` [6], thus resulting in slow transaction processing, wasted hard drive space, and long synchronization time. More seriously, the confidence of users in Ethereum will be shaken, and consequently the market price of Ethereum will be impacted [21]. Since each Ethereum node should maintain the complete copy of blockchain and replay all transactions in history for synchronization, such DoS attacks happened in history will also impact the newly enrolled nodes. Although Ethereum adjusted the gas costs of operations to defend against such known attacks [6,7], it remains unknown whether or not the new setting is resistant to unknown attacks and how to properly configure the gas costs of operations to mitigate DoS attacks.

In this paper, we make the *first* step to address this challenging issue by first proposing an emulation-based framework (in Sect. 4) to automatically measure the consumptions of computing resources of EVM operations. The framework consists of the interpretation handler for each EVM operation, the related data structures and diverse simulated environments in an attempt to explore all program paths of those handlers. The experimental result reveals that the latest setting in Ethereum is still not proper although it can mitigate the known DoS attacks. From this investigation, we obtain the insight that there may always

exist exploitable under-priced operations if the operation costs are fixed, because the factors influencing the costs of EVM operations keep changing.

Therefore, we propose a novel adaptive gas cost mechanism (in Sect. 5), which will dynamically adjust the costs of EVM operations according to the number of executions, to defend against known and unknown DoS attacks. This mechanism punishes the operations leading to abnormal high gas costs if they are executed much more frequently than before. Consequently, the exponentially increased gas costs will impede the attackers without unlimited money from conducting effective DoS attacks. Our experiments in a private blockchain show that the new mechanism can effectively thwart both known and unknown DoS attacks and introduce negligible additional gas consumption to benign users.

Moreover, by exploiting Ethereum's unique feature, we realize our mechanism through a novel approach in order to make it secure and flexible in terms of parameter adjustment. More precisely, we develop a specific smart contract and provide a patch to EVM. After patching the EVM, the developers of Ethereum can adjust the parameters by sending transactions to that smart contract, and then the updated EVM can fetch the parameters periodically by reading the storage of that smart contract. Our new approach leverages the underlying blockchain technique to make the parameters auditable and untamperable. Moreover, our approach has good deployability because it only needs updating the EVM once.

In summary, we make the following major contributions:

(1) We propose the first emulation-based measurement framework, which can automatically estimate the resource consumptions of EVM operations, to assess whether or not the gas costs in Ethereum are properly configured (Sect. 4).
(2) We propose a novel adaptive gas cost mechanism, which dynamically adjusts operation costs according to their execution times, to defend against known and unknown DoS attacks with negligible impacts on benign users.
(3) We design a new approach to realize our gas cost mechanism by exploiting Ethereum's smart contract and its underlying bloackchain technique. This approach makes our mechanism secure, flexible, easy to be deployed.
(4) We conduct experiments in a private blockchain to evaluate our mechanism. The results show that it can effectively thwart both known and unknown DoS attacks and introduce negligible additional gas consumption to benign users. Moreover, the parameters can be dynamically adjusted by authorized users.

The remainder of this paper is organized as follows. Section 2 introduces background knowledge. Section 3 presents our analysis of two real DoS attacks on Ethereum. Section 4 details the measurement framework. We describe the adaptive gas cost mechanism and its implementation in Sect. 5 and Sect. 6, respectively. The experiment results are introduced in Sect. 7. After summarizing related studies in Sect. 8, we conclude the paper in Sect. 9.

2 Background

This section introduces some background knowledge of Ethereum. Besides providing a cryptocurrency (i.e., Ether), Ethereum supports deploying and running smart contracts. There are two types of accounts in Ethereum, including external owned accounts (EOA) and smart contracts. The major difference between them is that only smart contracts contain executable bytecode [1]. Ethereum uses the underlying P2P overlay to deliver transactions among Ethereum nodes. A *transaction* refers to the signed data package that stores a message to be sent from an EOA to another account on the blockchain [1]. A block is a data structure to store zero or more transactions. Each node runs an Ethereum client that obeys Ethereum protocol [24]. The consensus in Ethereum is achieved by using a modified version of GHOST protocol [19], and as the result of the consensus, every node maintains the same copy of the blockchain. In particular, a newly joined node should download all blocks (i.e., synchronization) and then run all historical transactions to reach the same state as the other nodes.

Ethereum can be considered as a state machine where a state is a snapshot of the blockchain (e.g., the balances of all accounts, the value of a variable in a smart contract) and a transaction results in a state transfer. If the target of a transaction is a smart contract, the smart contract will be executed in EVM. Since EVM is usually embedded in the Ethereum client, the execution of smart contracts consumes the computing resources (e.g., CPU, disk, network) of each node. Consequently, a DoS attack will impact all nodes because each of them should execute all historical transactions. To prevent abusing computing resources, Ethereum leverages gas to charge execution fee from transaction senders. The amount of gas consumption is determined by the executed EVM operations, and different operations may have different gas costs [24]. In Sect. 3, we use real attacks to explain how attackers exploit the improper setting of gas cost to launch DoS attacks at low expense.

3 Analyzing Real DoS Attacks on Ethereum

This section dissects two real DoS attacks exploiting under-priced operations.

3.1 `EXTCODESIZE` Attack

Approach: The attacker sends lots of transactions to invoke a deployed smart contract involving many `EXTCODESIZE` operations, which gets the size of an account's code [24]. Such attack forces `EXTCODESIZE` to be executed roughly 50,000 times per block [7].

Symptom: Clients spend a very long time to process those blocks that contain the transactions sent from the attacker, and hence the throughput of Ethereum for processing transactions is decreased.

Cause: `EXTCODESIZE` has a very low gas cost (i.e., 20 in go-ethereum V 1.3.5), but it involves expensive operation (i.e., reading information from the disk). Hence, the execution of a great number of `EXTCODESIZE` results in busy I/O and slow transaction processing speed.

Countermeasure: New Ethereum (e.g., go-ethereum V 1.6) increases the gas cost of `EXTCODESIZE` to 700 [5] (in the source file gas_table.go). Consequently, the transaction senders have to pay 35 (= 700/20) times more money when using go-ethereum V 1.6. 700 gas is equal to about 0.000014 Ether (many senders set the gas price to 0.00000002 Ether at August, 2017), whose value is about 0.0042 USD (1 Ether can be exchanged into about 300 USD at August 13th, 2017 [11]). Although a single operation does not cost much, the accumulative gas consumption is considerable, because each transaction incurs the execution of many operations and there are more than 45 million transactions from the launch of Ethereum to August 13th, 2017 [2].

3.2 `SUICIDE` Attack

Approach: The attacker creates lots of smart contracts with a loop in their constructors. In the loop, the `SUICIDE` operation is executed. According to Ethereum's protocol, `SUICIDE` is used to remove the executed smart contract from the blockchain and send the remaining Ether to the designated account [24]. For each generated smart contract, the transaction for creating it triggers its constructor, and hence lots of `SUICIDE` whose target accounts do not exist, will be executed. Note that a nonexistent account does not need to be stored in the Ethereum state tree [6], which represents the state of the blockchain.

Symptom: About 19 million accounts were created by the attack, which consume considerable disk space, and thus the synchronization and transaction processing are slowed down.

Cause: If the target account does not exist, a `SUICIDE` operation will turn it into existent, which will be stored in the Ethereum state tree [6]. However, the gas cost of `SUICIDE` is zero. Therefore, an attacker creates a huge number of accounts by executing `SUICIDE` repeatedly at very low cost.

Countermeasure: New Ethereum increases the gas cost of `SUICIDE` to 5,000 and additional 25,000 if it creates a new account [5] (in the source file gas_table.go). Moreover, new clients can delete the zombie accounts created by the attack.

3.3 Remarks

From the above analysis, we learn that to exploit the under-priced operations for launching DoS attacks, the attacker has to first find or prepare a smart contract containing the under-priced operations, and then cause such operations to be

executed lots of times by sending transactions to the smart contract. Moreover, since the gas cost for sending a transaction is high (e.g., at least 21,000 in go-ethereum V 1.6), the attacker usually lets each transaction trigger multiple executions of the under-priced operations. To defend against such DoS attacks, we should either properly set the costs of EVM operations (i.e., remove under-priced operations) or force the attacker to pay a lot of money for executing the under-priced operations many times.

In Sect. 4, we propose a novel emulation-based measurement framework to assess whether or not the latest gas cost setting is proper. Unfortunately, we find that the latest setting still has exploitable under-priced operations, and it is difficult, if not impossible, to eliminate all under-priced operations if the operation costs are fixed, because the factors influencing the cost of each EVM operation keep changing. Therefore, we explore an alternative approach by proposing a novel adaptive gas cost mechanism in Sect. 5.

4 Emulation-Based Measurement Framework

Although Ethereum has changed the gas costs of some under-priced operations to defend against the known DoS attacks [6,7], little is known whether or not the latest gas cost setting is immune to DoS attacks exploiting under-priced operations. To address this issue, the resource consumption of each EVM operation should be measured. However, it is non-trivial to measure the computing resources consumed by a single EVM operation because the execution of a smart contract involves not only many EVM operations but also various utility functions for supporting the execution.

To tackle this problem, we propose a novel emulation-based measurement framework. More precisely, by exploiting EVM's architecture, we extract the interpretation handler for each operation (e.g., the *opAdd()* function is responsible for executing the addition operation), the related data structures (e.g., stack, memory, storage) from the EVM implementation, and prepare an emulated environment, which consists of the *Go* compiler, runtime libraries (e.g., the *bigInt* library to handle large integers) and the state of the blockchain (e.g., the balance of an account), for executing the operation. Then, we run the interpretation handler in the emulated environment millions of times, because a single run is too short to conduct the measurement, and record the execution time. Note that the current implementation of our framework can automatically measure the CPU consumption in terms of the execution time, and we will support the measurement of other resources in future work.

There is a challenge in preparing the emulated environment. In particular, since a handler may have various execution paths with different resource consumption, we need to explore all execution paths for measuring the handler's resource consumption. The example in Fig. 1 shows that the handler for executing `SUICIDE` consists of an expensive path (Line 15, *CreateStateObject()* allocates disk space to store accounts) if the target account is nonexistent since it will become existent after executing `SUICIDE` and a cheap path (Line 16)

```
1  func opSuicide(contract *Contract, stack *stack, ...){
2      balance := env.Db().GetBalance(contract)
3      env.Db().AddBalance(stack.pop(), balance)
4      env.Db().Delete(contract)
5  }
6  func (self *StateDB) AddBalance(addr common.Address, amount *big.Int) {
7      stateObject := self.GetOrNewStateObject(addr)
8      if stateObject != nil {
9          stateObject.AddBalance(amount)
10     }
11 }
12 func (self *StateDB) GetOrNewStateObject(addr common.Address) *StateObject {
13     stateObject := self.GetStateObject(addr)
14     if stateObject == nil || stateObject.deleted {
15         stateObject = self.CreateStateObject(addr) //a heavy path
16     }//else..., a cheap path
17     return stateObject
18 }
```

Fig. 1. An expensive path and a cheap path of *opSuicide*

if the target is existent. To address this challenge, we run the handler millions of times, providing with different inputs and proper runtime environment. If the operation manipulates the stack/memory/storage, we synthesize the stack/memory/storage with random length and generates random numbers as their items. If the operation needs the information from EVM (e.g., block number, gas price, gas limit) or the smart contract (e.g., code length, input to the contract, the address of the contract), we prepare an EVM/smart contract object with randomly generated fields. If the operation needs to interact with another account, we take into account the following three situations. First, if the target account is nonexistent, no special preparation is needed. Second, if the target account is an EOA, we generate one using the command provided by Etheruem's client. Third, if the target account is a smart contract, we develop and deploy one in the private chain, whose code is a `RETURN` since we measure the resources consumed by the invocation, rather than the execution of the invoked smart contract.

We classify all EVM operations into five categories in terms of the data structures on which they operate. The operations in the first category do not manipulate any data structures (e.g., `JUMPDEST`). The operations in the second category handle the stack (e.g., `ADD`). The operations in the third category get access to the specific fields related to blockchain (e.g., `ORIGIN`). The fourth category of operations manipulates the memory (e.g., `MSTORE`). The operations in the fifth category manipulate the storage (e.g., `SLOAD`). Note that in Ethereum *memory* is an infinitely expandable byte-array that resets after computation ends, while *storage* is a long-term key/value store that persists for the long term [19].

Figure 2 (the y-axis is on a log scale) presents the CPU consumptions of some EVM operations running 50 million times from all the five categories. Experiments are conducted on a desktop equipped with an Intel i3-4160 CPU and 8 GB memory. The number on top of each box is the operation's gas cost according to Ethereum's yellow paper [24]. All measurements repeat 100 times. `JUMPDEST` is the destination of a jump (e.g., `JUMP`, `JUMPI`) operation, which belongs to the first category. `ADD`, `SUB`, `MUL`, `DIV`, `SDIV`, `MOD`, `SMOD`, `ADDMOD`, `MULMOD` are arithmetic

operations. NOT and XOR are bitwise operations. ISZERO and LT are comparison operations. These operations belong the second category. ORIGIN is the representative of the third category which reads a field of the block's head. MSTORE and SHA3 belong to the fourth category. MSTORE writes a word to memory while SHA3 can operate multiple items in memory. In particular, SHA3 hashes the data in memory and its gas cost is the summation of basic gas (i.e., 30) with the gas for operating memory. The more memory it reads, the more gas it requires. EXP is a special arithmetic operation whose gas cost is the summation of basic gas (i.e., 10) with the remaining part which is determined by the bit length of the exponent. In other words, the gas cost of EXP becomes high if it has a large exponent. Figure 2 shows that EXP costs considerable CPU resources. SLOAD loads an item from the storage.

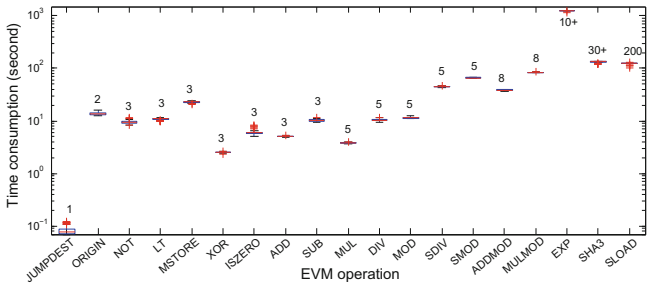


Fig. 2. Time consumptions of EVM operations

The results show that the latest gas costs are not proportional to the consumptions of CPU resources. For example, DIV (division) has the same gas cost of SDIV (signed division), but the execution time needed by DIV is about 23% (10.4 s/45 s) of that needed by SDIV. We find the reason by investigating the source code of handlers for DIV and SDIV, which is listed in Fig. 3. Figure 3 lists the source code (from go-ethereum V 1.6) of division (function *opDiv()*, Line 10) and signed division (function *opSdiv()*, Line 26), respectively, whose gas costs are equivalent. For the ease of presentation, we simplify the source code. We can see that the functions *opDiv()* and *opSdiv()* consist of stack operations (e.g., *stack.pop()*) and math computations (e.g. *x.And()*) provided by the *bigInt* library. Further experiments reveal that math computations (in red color) take up most of the execution time. We also find that the execution of a division operation needs 4 math computations (i.e., 1 *Div*, 1 *And*, 1 *Sub* and 1 *Exp*) at most whereas the execution of a signed division needs 11 (i.e., 3 *Sub*, 3 *Exp*, 2 *Abs*, 1 *Mul*, 1 *Div* and 1 *And*) at most. Hence, SDIV is more resource-consuming than DIV. Consequently, some operations (e.g., EXP, SHA3, as shown in Fig. 2) may be under-priced and thus could be exploited by DoS attacks.

```
1  func BigPow(a, b int64) {
2      r := big.NewInt(a)
3      return r.Exp(r, big.NewInt(b), nil)
4  }
5  tt256 = BigPow(2, 256)
6  tt256m1 = new(big.Int).Sub(tt256, big.NewInt(1))
7  func U256(x *big.Int){
8      return x.And(x, tt256m1)
9  }
10 func opDiv(stack *Stack){
11     x, y := stack.pop(), stack.pop()
12     if y.Sign() != 0 {
13         stack.push(math.U256(x.Div(x, y)))
14     } else {
15         stack.push(new(big.Int))
16     }
17 }
```

```
18 tt255 = BigPow(2, 255)
19 func S256(x *big.Int) {
20     if x.Cmp(tt255) < 0 {
21         return x
22     } else {
23         return new(big.Int).Sub(x, tt256)
24     }
25 }
26 func opSdiv(stack *Stack) {
27     x, y := math.S256(stack.pop()), math.S256(stack.pop())
28     If y.Sign() == 0 {
29         stack.push(new(big.Int))
30     } else {
31         if evm.interpreter.intPool.get().Mul(x, y).Sign() < 0 {
32             n.SetInt64(-1)
33         } else {
34             n.SetInt64(1)
35         }
36         res := x.Div(x.Abs(x), y.Abs(y))
37         res.Mul(res, n)
38         stack.push(math.U256(res))
39     }
40 }
```

Fig. 3. EVM source code for executing `DIV` and `SDIV` (Color figure online)

5 Adaptive Gas Cost Mechanism

The investigation in Sect. 4 shows that it is not easy to properly assign gas costs to EVM operations. Hence, we propose a novel adaptive gas cost mechanism for defending against DoS attacks.

5.1 Threat Model

We assume that the attacker can discover under-priced operations (if any) and then launch the attack by invoking either existing smart contracts or new smart contracts crafted by the attacker. Moreover, the attacker is rational and does not have unlimited money for launching attacks. In this case, she will give up the attack if her money cannot force the under-priced operations to be executed for lots of times. Moreover, she will not send a transaction that can execute the under-priced operations only a few times because sending a transaction is not cheap (i.e., gas cost is 21,000). Last but not least, normal users will not accept a gas cost mechanism that charges much money from them.

5.2 Adaptive Adjustment of Gas Costs

Exploiting the observation in Sect. 3.3 that a successful DoS attack has to trigger lots of executions of under-priced operations, we propose a new mechanism that increases the gas cost of an operation dynamically if it has been executed much more frequently than before. More precisely, we collect the execution traces (i.e., a sequence of executed operations) of normal transactions, and model the execution frequency of each EVM operation. Then, for every new transaction, we set a basic gas cost for each operation by default, and count the number of executions of each operation. If the number of an operation is larger than a threshold, its gas cost will be increased. The advantage of our mechanism is that it does *not* need to know which operations are under-priced. Instead, it punishes

the over-frequent EVM operation through the increased gas cost. Hence, it can defend against known and unknown DoS attacks.

We define a threshold μ_i for the operation i as shown in Eq. (1). The operation i that has been executed for more than μ_i in *one* transaction is regarded as over-frequent, and its gas cost will be increased. ave_i and std_i stand for the average and the standard deviation of the number of executing operation i, respectively. Section 6.1 details how to compute them. *base_count* is an integer used to prevent increasing the gas cost of an infrequently-executed operation too fast. m is a parameter for adjusting the threshold.

$$\mu_i = max\{base\_count, ave_i + m \times std_i\} \tag{1}$$

The gas cost of an EVM operation is dynamically adjusted according to Eq. (2), where $count_i$ is the number of executions of operation i, $base\_gas_i$ is the default gas cost of i. We uses an exponential function to punish over-frequent operations with accelerating increments in gas costs. Its base (i.e. $\alpha > 1$) determines the speed of increasing the gas cost. We let the exponent as $\frac{count_i}{\mu_i} - 1$ that includes μ_i for taking into account the operation's normal frequency. Since our mechanism will assign an operation a very high gas cost if it has been executed much more times in a transaction than before, it deters an attacker from executing an under-priced operation many times by one transaction. Moreover, our mechanism avoids charging much more gas from benign senders by setting proper parameters. We evaluate the effects of various parameters in Sect. 7.3. gas_i is restored to $base\_gas_i$ for a new transaction, and hence the attacker cannot affect the initial operation costs of benign transactions. Figure 4 shows the curves of Eq. (2) with various parameters, indicating that μ_i and α can affect the point from where to increase gas cost and the speed to increase gas cost, respectively. We have several observations. First, μ_i determines the point from where gas_i should be increased. Moreover, α determines the increasing speed of gas_i. Typically, gas_i should be increased with the increase of execution number $count_i$, and hence α should be larger than 1.

$$gas_i = \begin{cases} base\_gas_i, & if\ count_i \leq \mu_i \\ base\_gas_i + \alpha^{\frac{count_i}{\mu_i} - 1}, & if\ count_i > \mu_i \end{cases} \tag{2}$$

Section 5.3 will describe the way to adjust the parameters in Eqs. (1) and (2), and we will try other functions (e.g., linear, polynomial) in Eq. (2) in future work.

5.3 Dynamic Parameter Configuration

Since Ethereum and its smart contracts evolve over time, the parameters should be changed accordingly. Therefore, we need an approach for dynamic parameter configuration. This approach should meet the following requirements. First, the parameter configuration should be auditable by any users of Ethereum. Second, the parameter configuration should be secure so that attackers cannot modify the parameters. Third, the approach should not need to frequently update Ethereum client due to the risk of hard fork.

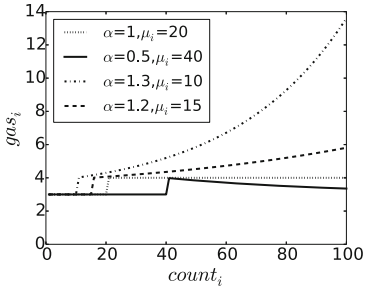


Fig. 4. Curves of Eq. (2), $base\_gas_i = 3$

Exploiting Ethereum's unique feature, we propose a novel approach for realizing dynamic parameter configuration by developing a specific smart contract and providing a patch to EVM. The developers of Ethereum can adjust the parameters by sending transactions carrying new parameters to that smart contract. They can adjust a variable, *block number*, in the smart contract, which is used to determine when the new setting takes effect. Then, the patched EVM can fetch the parameters periodically by reading the storage of that smart contract. The period (measured by blocks) of querying new parameters should be shorter than the difference between the variable, block number, in the smart contract and the block number when setting the new parameters so that all clients can get the newest setting before the block when the setting takes effect.

Our new approach leverages the underlying blockchain technique to make the parameters auditable and untamperable. Note that no one can change the setting of gas costs by just subverting her EVM. Moreover, the smart contract for updating parameters cannot be tampered by attackers who do not have more than 50% computing power because the contract itself will be validated in the process of consensus. The change of parameters will be auditable because all transactions are publicly available in the blockchain. Last but not least the Ethereum client (i.e., its EVM) should only be updated once for adopting our new gas cost mechanism. After that, they do not need to be updated again for using the new parameters.

6 Implementation

The implementation of our new mechanism consists of four parts (Fig. 5). The first part collects execution traces of smart contracts and computes ave_i and std_i. Part 2 is the smart contract storing the parameters that can be updated by Ethereum developers. Part 3 and 4 describe the patch to EVM, including how to fetch new parameters and how to apply them, respectively.

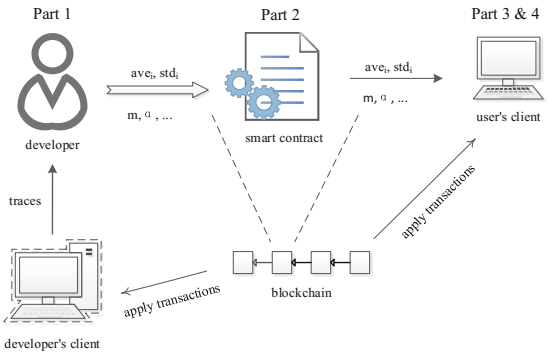
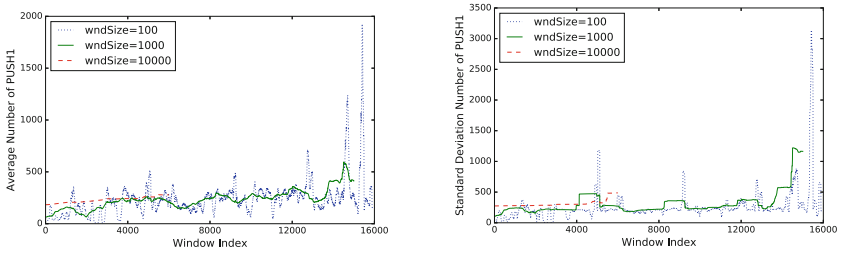


Fig. 5. Overview of our implementation.

6.1 Computing ave_i and std_i

To compute ave_i and std_i, we first leverage EVM's built-in tracing ability to record all execution traces. We define a sliding window, and use all traces within that window for computing ave_i and std_i. Figure 6 shows ave_i and std_i of `PUSH1` with different window sizes (i.e., 100, 1,000 and 10,000) in the first 16,000 execution traces since the launch of Ethereum. We assume that these traces were triggered by benign transactions since no known attacks were discovered in them. Please note that `PUSH1` is the most frequent operation, which pushes one byte on stack.



(a) `Ave` (b) `Std`

Fig. 6. Average number and standard deviation of the executions of `PUSH1`. (Color figure online)

The x-axis gives the window index and for example, a point (x, y) on the red line of Fig. 6(a) indicates that ave_i of `PUSH1` of the traces within the window $[x + 1, x + 10,000]$ is y. We can see that the ave_i of `PUSH1` increases as time goes on, indicating that smart contracts become more complicated than before. Second, as we expected, the larger the window is, the more stable ave_i and std_i will be. Moreover, it is difficult for an attacker to tamper ave_i and std_i by filling the large window with crafted transactions. Our approach allows developers to adjust the window size.

6.2 Smart Contract

We implement a smart contract (as shown in Fig. 7) to store parameters which allows the contract's creator to update parameters through executing transactions, and then we deploy it on our private blockchain. For ease of presentation, we omit the details of updating ave_i and std_i of each operation i, which is the same as the updating the other parameters (e.g., m). Line 2 declares several global variables which store in the storage. *address* (Line 3) is a built-in variable type of Ethereum which can only be used for storing account address. N is the time interval of two consecutive queries, and *delta* is a small number that we consider all clients can get the new setting in the time period of $N + delta$ (Line 13). The function *AdaptiveGas()* is the construct function that will be executed during the creation of the smart contract. Please note that the arguments of *AdaptiveGas()* are also given in the transaction for contract creation.

```
1  contract AdaptiveGas {
2      uint base_count, m, alpha, blk_num, N, delta;
3      address creator; //owner
4      function AdaptiveGas(uint init_base_count, uint init_m, uint init_alpha,
           uint init_blk_num, uint default_N, uint default_delta){
5          base_count = init_base_count; m = init_m;
6          alpha = init_alpha; blk_num = init_blk_num;
7          N = default_N; delta = default_delta;
8          creator = msg.sender; //set contract owner
9      }
10     function UpdateSetting(uint new_base_count, uint new_m,
           uint new_alpha, uint new_blk_num){
11         require(msg.sender == creator); //authentication
12         if (new_blk_num < block.number + N + delta)
13             new_blk_num = block.number + N + delta;
14         base_count = new_base_count; m = new_m;
15         alpha = new_alpha; blk_num = new_blk_num;
16     }
17 }
```

Fig. 7. The smart contract for updating the setting of parameters

Besides setting the default parameters in *AdaptiveGas()* (Lines 5–7), we record the contract owner (Line 8), ensuring that only the owner can change parameters setting (Line 11). The function *UpdateSetting()* accepts the new setting of parameters from transaction senders. Lines 12, 13 ensure that the time period ($N + delta$) is enough for all clients to check the update. Please note that *msg.sender* and *block.number* are two built-in properties of Ethereum that get the address of transaction sender and the number of block which contains the transaction, respectively. Please note that the transaction fees for sending the transactions to adjust paramters are negligible for Ethereum official society because a single transaction does not cost much (always less than 1 USD [2]) and parameters do not need to adjust very frequently.

6.3 Querying New Parameters

Figure 8 shows the code snippet (simplified for presentation) for an EVM to get the new setting of parameters. Since each Ethereum node keeps a complete copy

of blockchain, their EVMs can get the values of all storage variables given the address of the smart contract by accessing the local copy of blockchain. It is more efficient than an intuitive approach that fetches the new parameters by sending a transaction to the smart contract, because the latter will add transactions to the blockchain periodically and cause additional fee for sending transactions. Our approach can avoid these issues. Line 1 specifies the address (i.e., *ac43...*) of the contract, which is known because the contract is developed and deployed by us. Then, Lines 2–5 obtain individual parameters by directly accessing (i.e., invoking the internal function *evm.StateDB.GetState()* of EVM) the storage of the contract. The integers 0, 1, etc. give the locations of parameters stored in the storage. Finally, those parameters are used for computing gas costs.

```
1  var contract = common.HexToAddress("ac43...")
2  base_count = evm.StateDB.GetState(contract, 0)
3  m = evm.StateDB.GetState(contract, 1)
4  alpha = evm.StateDB.GetState(contract, 2)
5  blk_num = evm.StateDB.GetState(contract, 3)
```

Fig. 8. Modifications of EVM to obtain new parameters.

6.4 Applying New Parameters

We modify go-ethereum V 1.3.5 to realize our mechanism because it has several known under-priced operations, and we compare the original V 1.3.5 with the patched one in Sect. 7.1. When Ethereum starts, we load the setting of parameters (e.g., ave_i, std_i, m, α) in the entry function (i.e., the *main()* function in \cmd\geth\main.go). Please note that the default gas cost of each operation (i.e., $base\_gas_i$) is the same as that in go-ethereum V 1.3.5. We replace the code in the function *CalculateGasAndSize()* in \core\vm\vm.go, which is responsible for computing the gas consumption of individual operation, with our code to calculate gas cost and increase the execution number of the EVM operation by one. In other words, Eqs. (1) and (2) are implemented in *CalculateGasAndSize()*. The number of executions will be reset before the execution of each transaction, which is implemented in the function *ApplyTransaction()* in \core\state_processor.go. To reduce the runtime overhead, we cache the gas costs of operations, which have already been computed, in main memory.

7 Evaluation

This section answers the following questions through experiments.

RQ1: Can our mechanism thwart known and unknown DoS attacks effectively?
RQ2: How much additional gas will be charged from benign users by our mechanism?
RQ3: What are the effects of parameters?

All experiments are conducted in a private Ethereum blockchain on a desktop equipped with an Intel Xeon E312 processor and 8 GB memory. Our private blockchain has one miner and is isolated with the public Ethereum blockchain and other testing blockchains. We create an account to hold the rewards from mining. We guarantee that the account has enough money to send transactions by setting a low mining difficulty. Every block also has a gas limit, dubbed BGL (Block Gas Limit), which restricts the size of a block (i.e., the number of transactions contained in the block). The BGL is set as 4 million, which is comparable with that in the public chain at present. We let the TGL be equal to the BGL, in order to see how many under-priced operations can be executed by a single transaction using the original gas cost mechanism and our mechanism, respectively. The parameters *base_count*, m and α in Eqs. (1) and (2) are set to 5, 3 and 2 by default, respectively. We evaluate our mechanism under different settings in Sect. 7.3.

7.1 Experiments with DoS Attacks

We simulate the two real attacks [6,7] in our private blockchain. To launch the `EXTCODESIZE` attack, we develop a smart contract with a public function *extAttack()* that can be called by our account. *extAttack()* has a loop where we use inline assembly to execute `EXTCODESIZE` directly. The `SUICIDE` attack is launched in a more intricate way since a smart contract will be removed (i.e., cannot get accessed) by executing `SUICIDE`. The `SUICIDE` attack exploits the feature of Ethereum: a smart contract will not be removed before the completion of the transaction that triggers the `SUICIDE` operation. Consequently, we create a smart contract whose constructor invokes `SUICIDE` in a loop. When creating the contract, the corresponding transaction executes `SUICIDE` repeatedly. We also use the built-in tracing ability of EVM to record the execution traces of smart contracts as well as the gas consumption of each operation.

The experimental results reveal that the two attacks execute 92,494 and 11,335 times of `EXTCODESIZE` and `SUICIDE`, respectively, in one transaction using the original (i.e., go-ethereum V 1.3.5) gas cost mechanism. By contrast, the two attacks only execute 99 and 81 times of `EXTCODESIZE` and `SUICIDE`, respectively, with the same cost (i.e., 4 million gas) after our mechanism is applied. Figure 10(a) (y-axis is on a log scale) and Fig. 10(b) shows the gas cost of each operation in descending order after the attacks when our mechanism used. Note that these two figures do not include all operations due to the page limit. We can see that the gas costs of the two under-priced operations become very expensive (i.e., 457,119 and 37,640 respectively) after attacks. We also find some other expensive operations (e.g., `CALLDATALOAD`, `CALL`) because they are also in the loop, resulting in over-frequent executions than before. Figure 9 demonstrates that the execution frequencies of different operations vary. Moreover, the two under-priced operations (i.e., `EXTCODESIZE` and `SUCIDE`) exploited by real attacks are rarely executed by benign users.

To evaluate our approach against unknown DoS attacks, we synthesize three attacks by executing three under-priced operations (i.e., `EXTCODECOPY`, `SLOAD`

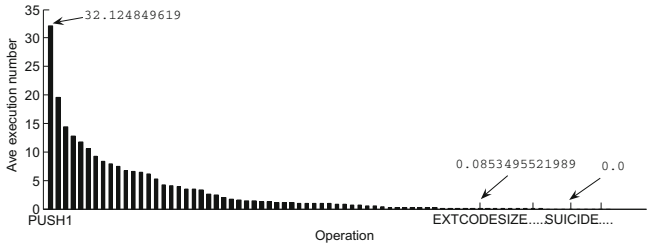
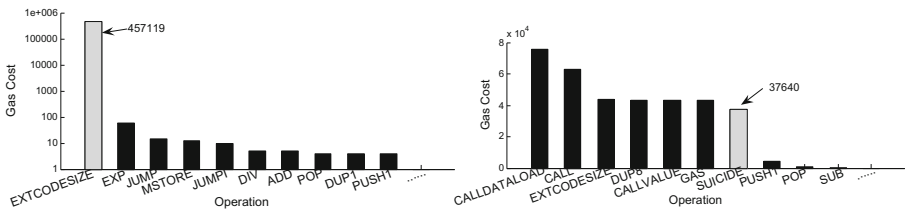


Fig. 9. Average execution number of every EVM operation, 10,000 benign transactions collected from 07:40:00 AM, April 28, 2017 to 01:58:56 PM, April 28, 2017



(a) `EXTCODESIZE` (b) `SUICIDE`

Fig. 10. Gas of each operation after attacking

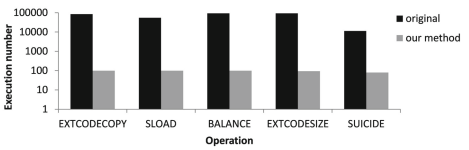


Fig. 11. Execution numbers of under-priced operations

and `BALANCE`) in a loop, respectively, which are similar to the `EXTCODESIZE` attack. Note that go-ethereum V 1.3.5 will be affected by the DoS attacks exploiting these operations whereas the latest version of Ethereum has increased their gas costs. Figure 11 demonstrates that our method reduces the number of executions of under-priced operations by several orders of magnitude. Therefore, the answer to RQ1 is:

Our gas cost mechanism can effectively thwart known and unknown DoS attacks.

7.2 Experiments with Normal Transactions

To evaluate how much additional gas will be charged from normal users by our mechanism, we first randomly select 10 smart contracts and then replay their transactions in the original go-ethereum V 1.3.5 and the updated go-ethereum

V 1.3.5 with our gas cost mechanism, respectively. 84 transactions in total are replayed, and 15 transactions (2 to one contract and the other 13 to another contract) out of them incur additional gas by our mechanism. The total gas consumed by 84 transactions under the original gas cost setting is 2,441,340, and the total additional gas incurred by our mechanism is 444. Therefore, the percentage of additional gas charged from benign users is about 0.018%.

As a case study, we detail the experiment with one smart contract. More precisely, the smart contract is deployed at 0x61F9d1cE56aC1623FeD4e949 D7D420251fef0896. We compile the source and deploy the smart contract in the private blockchain. There are 37 transactions to that smart contract in total until April 29, 2017. We do not replay the transaction for contract creation since it does not trigger the execution of any public functions provided by the smart contract, nor the 4 transactions with internal transactions because our private blockchain is isolated from other accounts. Note that an internal transaction is not a real transaction and will not be stored in the blockchain. Instead, it is made by calling (via `CALL`, `CALLCODE`, `DELEGATECALL` etc.) an account from a smart contract. We also skip the transaction running out of gas, and hence we replay 31 $(37 - 1 - 4 - 1)$ transactions.

The results show that 18 out of 31 transactions consume the same amount of gas under our mechanism and the original mechanism. The total increment in gas consumption of the other 13 transactions incurred by our mechanism is 130, and the largest increment in gas consumption of one transaction is 10. Please note that the total gas consumption of the 31 transactions under original mechanism is 1,357,654. That is, the increment in gas consumption due to our mechanism is negligible (i.e., 0.01%). Hence, the answer to RQ2 is:

Our gas cost mechanism charges negligible additional gas from benign users.

7.3 Different Parameter Settings

We evaluate our mechanism under three different settings as listed in Table 1. For example, "3(5/1.2)" means that in setting 3, m and α are set to 5 and 1.2, respectively. Please note that the setting 2 is the default setting. Table 1 also presents the execution numbers of under-priced operations and the highest gas costs of them. For example, "48/1,026,690" in row 2, column 2 indicates `EXTCODECOPY` executes 48 times under setting 1 and the gas cost of the 48th `EXTCODECOPY` is 1,026,690. Please note that the gas cost of an operation keeps increasing if its execution number exceeds μ_i (Eq. (2)).

The experimental results demonstrate that our approach is sensitive to DoS attacks by setting a small m and a large α. The setting 1 detects attacks quicker (i.e., the execution numbers of under-priced operations are the lowest) than the other two settings. For example, the `EXTCODESIZE` attack executes 48 `EXTCODESIZE`, and its gas cost reaches 1,026,187 under the setting 1 whereas the attack executes 328 `EXTCODESIZE` and the gas cost of `EXTCODESIZE` reaches 131,049 under the setting 3. The results are as expected since the threshold μ_i depends on m and α determines the speed of increasing gas costs.

Table 1. Execution numbers (before/) and the highest gas costs (after/) of under-priced operations under different settings

Setting	BALANCE	EXTCODECOPY	EXTCODESIZE	SLOAD	SUICIDE
1(1/5)	48/1,026,387	48/1,026,690	48/1,026,687	48/1,026,187	22/237
2(3/2)	99/456,819	99/457,122	99/457,119	99/456,619	81/37,640
3(5/1.2)	329/135,590	328/131,052	328/131,049	329/135,390	289/31,440

One may feel strange that SUICIDE presents different trend with the other attacks under different settings. For example, the gas cost of SUICIDE under the setting 2 is larger than that under the other two settings, whereas the gas costs of the other four under-priced operations under the setting 1 reach the largest value. The reason is that SUICIDE is not the most expensive operation during attack (as shown in Fig. 10(b)), and thus the execution number of SUICIDE is influenced by the gas consumption of other expensive operations. Figure 12 shows the increment in gas consumption of applying 31 transactions to the smart contract at 0x61F9d1cE56aC1623FeD4e949D7D420251fef0896 under three different settings. The x-axis specifies transactions in short, e.g., *3d1b* is the first two bytes of a transaction hash which is 32 bytes in length. The results reveal that a setting that is more sensitive to DoS attacks may charge more execution fee from benign users.

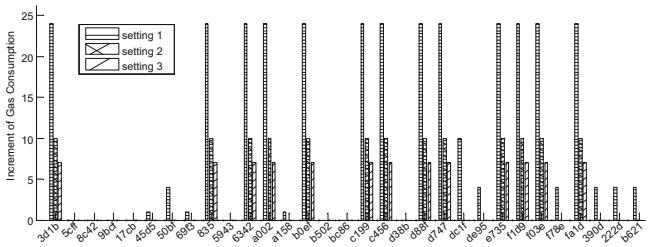


Fig. 12. Additional gas consumption of 31 transactions under three different settings

We also evaluate whether our mechanism can defend against DoS attacks exploiting the five under-priced operations under the default setting with different window sizes. We compute ave_i and std_i of each operation i for different windows sizes, including 100, 500, 1,000, 5,000 and 10,000. We use the first 16,000 transactions since the launch of Ethereum for experiments, which do not include attacks. The attacks exploiting the five under-priced operations are conducted in our private chain for this experiment.

Figure 13 presents the execution numbers of SLOAD with different window sizes (the experiments of other four under-priced operations produce similar results). It shows that our method is effective using the parameters computed

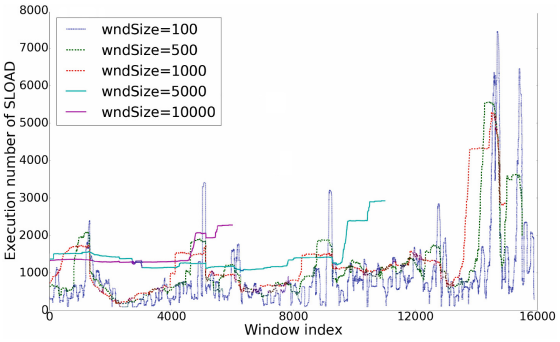


Fig. 13. Execution numbers of `SLOAD` with different window sizes

from all window sizes because the under-priced operation executes more than the threshold μ_i at any window sizes and hence its gas cost increases during attacks. More precisely, the original gas cost method allows `SLOAD` to execute nearly 100,000 times (Fig. 11) whereas our method reduces this number significantly.

We assume all transaction in the windows for computing parameters are benign. Attackers may want to place crafted transactions into the windows to affect the process of computing ave_i and std_i for the sake of evading the detection. To make our approach more robust, we suggest analysts to set a relatively large window size (e.g., 10,000) that consists of many transactions. In another words, a large window size raises the difficulty for attackers to fill the window with crafted transactions and tamper parameters. Besides, after detecting an attacking transaction, we can filter out the attacking transactions in the windows by matching the transaction senders, attached data (specifying which function to call and providing augments) of transactions, the execution traces of contracts etc.

Hence, the answer to RQ3 is:

The experimental result show that DoS attacks can be detected quickly and negligible additional gas is introduced to benign users under different parameter settings. Our mechanism allows developers to easily adjust parameters with the evolving of Ethereum.

8 Related Work

DoS attacks have posed a severe threat to the Internet [17,25] and various systems [10,14] and services [23]. Although DoS attacks on Ethereum have been reported, there lacks of a systematic study on the attacks and the defense mechanisms. To the best of our knowledge, this paper presents the first work on defending against under-priced DoS attacks on Ethereum.

BLOCKBENCH [12] is an evaluation framework for measuring the throughput, latency, scalability and fault-tolerance of private blockchains. Yasaweerasinghelage et al. propose to predict the latency of blockchain-based applications using architectural performance modeling and simulation tools [26]. However, they [12,26] do not investigate the consumptions of computing resources for executing EVM operations. OYENTE [18] is a symbolic executor for smart contracts which discovers four types of security vulnerabilities. GASPER [9], based on OYENTE, finds under-optimized smart contracts automatically that cost more gas than necessary. A recent survey [3] reports that smart contracts suffer from several kinds of vulnerabilities. One kind is *gasless send*, indicating that a transaction sender may not consider the situation that sending Ether to another account is possible to fail due to the out-of-gas exception. Sergey et al. reveal that smart contracts will suffer from similar problems that often occur in transitional concurrent programs [22]. However, they [3,9,18,22] do not consider DoS attacks to Ethereum, which exploit under-priced EVM operations.

Verification is used for verifying the runtime safety and functional correctness of smart contracts. Bhargavan et al. propose to translate a smart contract into F*, a functional language before formal analysis [4]. Similarly, Pettersson and Edström suggest developing smart contracts in Idris, a functional language, and using type system to capture errors at compile time [20]. Hirai formally defines EVM in Lem, an intermediate language similar to a functional language, facilitating further analysis and generation of smart contracts [13]. However, they neither verify nor detect DoS attacks due to under-priced operations [4,13,20]. Hawk is a smart contract system protecting transactional privacy [16]. Town Crier [27] aims at providing trustworthy data to smart contracts since they need data out from the blockchain. Juels et al. report that smart contracts can be used to commit crimes, such as privacy leakage, theft of cryptographic keys [15]. However, they [15,16,27] do not discuss the threats resulting from Ethereum DoS attacks.

9 Conclusion

We investigate the gas cost setting in Ethereum because it could be exploited to launch DoS attacks. By proposing an emulation-based framework to automatically measure the resource consumptions of EVM operations, we find that Ethereum does not assign proper gas costs to operations and it is difficult to properly assign fixed gas costs to operations for defending against known and unknown DoS attacks. Therefore, we propose a DoS-resistant gas cost mechanism, which dynamically adjusts the costs of operations according to the number of executions. Our approach is flexible and secure, and we design a special smart contract that collaborates with the customized EVM to avoid frequently updating Ethereum client. Experimental results show that our method effectively thwarts known and unknown DoS attacks, and introduces negligible additional gas to benign users.

Acknowledgment. This work was supported in part by the National Natural Science Foundation of China, No. 61402080, No. 61572115, No. 61502086, No. 61572109, China Postdoctoral Science Foundation founded project, No. 2014M562307, and Shenzhen City Science and Technology R&D Fund (No. JCYJ20150630115257892).

References

1. Ethereum homestead documentation (2017). https://goo.gl/V6PmCg
2. Etherscan - transactions (2017). https://etherscan.io/txs
3. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts (SoK). In: Proceedings of the POST (2017)
4. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanélla-Beguelin, S.: Formal verification of smart contracts: short paper. In: Workshop, PLAS (2016)
5. Buterin, V.: Eip150: long-term gas cost changes for IO-heavy operations to mitigate transaction spam attacks (2016). https://goo.gl/8gwNCL
6. Buterin, V.: A state clearing faq (2016). https://goo.gl/x5QRrd
7. Buterin, V.: Transaction spam attack: next steps (2016). https://goo.gl/uKi9Ug
8. Carter, J.: Bitcoin vs distributed ledger vs ethereum vs blockchain (2016). https://goo.gl/3EQVdJ
9. Chen, T., Li, X., Luo, X., Zhang, X.: Under-optimized smart contracts devour your money. In: Proceedings of the SANER (2017)
10. Chen, T., Li, X., Luo, X., Zhang, X.: System-level attacks against android by exploiting asynchronous programming. Softw. Qual. J. 1–26 (2017). https://doi.org/10.1007/s11219-017-9374-6
11. CoinGecko: Ethereum/us dollar price chart (2017). https://goo.gl/pezZAn
12. Dinh, T., Wang, J., Chen, G., Liu, R., Ooi, B., Tan, K.: Blockbench: a framework for analyzing private blockchains. In: Conference on SIGMOD/PODS (2017)
13. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Proceedings of the WTSC (2017)
14. Jiang, M., Wang, C., Luo, X., Miu, M., Chen, T.: Characterizing the impacts of application layer DDoS attacks. In: Proceedings of the IEEE ICWS (2017)
15. Juels, A., Kosba, A., Shi, E.: The ring of Gyges: investigating the future of criminal smart contracts. In: Proceedings of the CCS (2016)
16. Kosba, A., Miller, A., Shi, E., Wen, Z., Papamanthou, C.: Hawk: the blockchain model of cryptography and privacy-preserving smart contracts. In: Proceedings of the S&P (2016)
17. Luo, X., Chang, R.: Optimizing the pulsing denial-of-service attacks. In: Proceedings of the DSN (2005)
18. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the CCS (2016)
19. Maltsev, P.: White paper: a next-generation smart contract and decentralized application platform (2017). https://goo.gl/6Y8ivs
20. Pettersson, J., Edström, R.: Safer smart contracts through type-driven development. Master's thesis, Chalmers University Of Technology And University Of Gothenburg (2016)
21. Rocky: Ethereum faces another dos attack (2016). https://goo.gl/sAUjJ7
22. Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: Proceedings of the WTSC (2017)

23. Tang, Y., Luo, X., Hui, Q., Chang, R.: Modeling the vulnerability of feedback-control based internet services to low-rate dos attacks. IEEE Trans. Inf. Forensics Secur. **9**(3), 339–353 (2014)
24. Wood, G.: Ethereum: a secure decentralised generalised transaction ledger, EIP-150 revision (2016). http://gavwood.com/paper.pdf
25. Xue, L., Luo, X., Chan, E., Zhan, X.: Towards detecting target link flooding attack. In: Proceedings of the USENIX LISA (2014)
26. Yasaweerasinghelage, R., Staples, M., Weber, I.: Predicting latency of blockchain-based systems using architectural modelling and simulation. In: Conference on ICSA (2017)
27. Zhang, F., Cecchetti, E., Croman, K., Juels, A., Shi, E.: Town crier: an authenticated data feed for smart contracts. In: Proceedings of the CCS (2016)