GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts

Ting Chen, Youzheng Feng, Zihao Li, Hao Zhou, Xiapu Luo‡, Xiaoqi Li, Xiuzhuo Xiao, Jiachi Chen, and Xiaosong Zhang

Abstract—Ethereum, the largest blockchain for running smart contracts, charges the people who send transactions to deploy or invoke smart contracts for thwarting resource abuse. The amount of transaction fee depends on the size of that contract and the operations executed by that contract. Consequently, smart contracts with inefficient code will waste money. In this paper, we propose and develop the first tool, named `GasChecker`, for automatically identifying gas-inefficient code in smart contracts, and conduct the first empirical study on the prevalence of gas-inefficient code in the deployed smart contracts. More precisely, we first summarize ten gas-inefficient programming patterns and propose a new approach based on symbolic execution (SE) to detect them in the bytecode of smart contracts. To make our approach scalable to analyze millions of smart contracts, we parallelize SE by tailoring it to the MapReduce programming model, and propose a new feedback-based load balancing strategy to effectively utilize cloud resources. Extensive experiments show that `GasChecker` scales well with the increase of workers. The empirical study demonstrates that lots of real smart contracts contain various inefficient code. Manual investigation demonstrates that only 2.5% of discovered gas-inefficient instances are false positives.

Index Terms—Smart contract, scalable analysis, gas-inefficient pattern, symbolic execution, parallelization, MapReduce

1 Introduction

The supporting of smart contracts is the landmark of blockchain 2.0, which allows running various applications on the blockchain other than cryptocurrency (e.g., bitcoin) [1]. A *smart contract* is an autonomous computer program that, once started, executes automatically and mandatorily according to the program logic defined beforehand [2]. Due to many unique advantages (e.g., automatic execution in a deterministic manner, without trusted intermediaries, highly resistant to forgery), smart contracts have the power to reshape a number of industries, in which retail banking, insurance, financial exchange, marketplaces, and content platforms are the most impacted [3]. Therefore, many blockchain systems support smart contracts, such as Ethereum, Counterparty, Stellar, Monax, Lisk [4]. Among them, Ethereum is the largest one with more than 10 billion USD market capitalization[1], 8 million smart contracts and 200 million transactions[2] until September 2019. Moreover, smart contract developers are in great demand [5]. Therefore, this study concentrates on smart contracts of Ethereum. The terms blockchain and smart contract refer to Ethereum blockchain and Ethereum smart contract hereinafter if not special specified.

Smart contracts are typically developed in a high-level language (e.g., Solidity) and then compiled into a special bytecode form, the EVM bytecode, which can be executed in the Ethereum Virtual Machine (EVM), the runtime system of Ethereum. According to Ethereum's specification [6], every node in Ethereum maintains a complete copy of blockchain, and will replay all transactions stored in the blockchain. In particular, a smart contract will be executed if it is the recipient of a transaction. Since every node needs to run all smart contracts in the bytecode format using EVM, the execution of smart contracts consumes the computing resources of every node. To prevent resource abuse, especially Denial-of-Service (DoS) attacks that exhaust the computing resources (e.g., CPU, disk, network) of nodes [7], Ethereum adopts the *gas* mechanism that charges the transaction senders. In other words, transaction senders have to pay for deploying or invoking smart contracts. The amount of money should be charged for deploying and invoking a smart contract depends on the size of the deployed smart contract and the operations executed in that smart contract, respectively.

We name a smart contract that costs more gas than necessary as a *gas-inefficient smart contract*. Such contracts will waste a lot of gas because gas-inefficient patterns are prevalent in deployed smart contracts (Section 6.3), which could be invoked an unlimited number of times. Many factors can result in gas-inefficient smart contracts, e.g., unawareness of gas waste, inexperienced developers, insufficient compiler optimizations, the absence of auxiliary tools.

In this paper, we first identify ten gas-inefficient patterns, which overcharge the developers and users of smart contracts. Then, we design and develop `GasChecker`, a novel system to automatically detect these patterns in the bytecode of smart contracts. `GasChecker` does not need source code, because the proportion of open-source smart contracts is less than 1% [8]. To make `GasChecker` scalable to process millions of smart contracts, we propose a new approach to parallelize SE by leveraging cloud computing platform.In particular, we tailor parallel SE into MapReduce programming model and propose a feedback-based load balancing strategy to improve the utilization of computing resources. The experimental results show that `GasChecker` has high precision and low false positive rate (i.e., 2.5%) (Section 6.2). Moreover, `GasChecker` scales well with the increase of workers, and our proposed load balancing strat-

‡ *The corresponding author.*

1. CryptoCurrency Market Capitalizations: https://coinmarketcap.com/.

2. Etherscan: https://etherscan.io/.

egy is effective (Section 5). By applying GasChecker to the deployed smart contracts, we observe that these deployed smart contracts contain lots of gas-inefficient code (Section 6.3) and some gas-inefficient patterns occur frequently. Unfortunately, recent compilers cannot eliminate gas-inefficient patterns, although new compilers outperform old compilers in optimizing gas-inefficient code (Section 6.4). Moreover, we find that much money can be saved after optimizing gas-inefficient code. For example, by optimizing three gas-inefficient patterns in 1,500 smart contracts, we can save 1,520 USD (Section 6.5).

Our position paper first points out the issue of gas-inefficient smart contracts [9]. This full paper extensively extends our previous work from four aspects. First, we identify ten gas-inefficient programming patterns whereas our position paper introduces seven patterns. Note that this work adds four new patterns, and excludes one pattern proposed in our position paper because our experiments show that such pattern is not appeared in practice. Second, we design and develop a new system named GasChecker for detecting all ten patterns automatically in EVM bytecode whereas only three patterns could be detected by our previous work [9]. Third, to make GasChecker scalable to handle millions of smart contracts, we propose a new approach to parallelize symbolic execution (SE) by leveraging cloud computing platform whereas sequential SE is used in our previous work [9]. Finally, we conduct much more experiments in this paper than the position paper, e.g., scalability, detection precision of GasChecker, amount of saved money by optimizing contracts.

Our previous tool, PACCI is the *first* work to parallel SE via MapReduce [10]. However, GasChecker is implemented from scratch rather than reuse PACCI for two reasons. First, GasChecker analyzes EVM bytecode while PACCI handles binaries. Second, PACCI does not consider load balancing.

This study can benefit developers in several domains. First, the developers of smart contracts can avoid the gas-inefficient patterns identified in this paper. Second, compiler developers can improve compilers by replacing gas-inefficient code with efficient code during compiling. Third, the developers of Ethereum can create a JIT compiler and embed it into the runtime system (i.e., EVM) for optimizing gas-inefficient code at runtime. Last but not least, this work may motivate a new optimization service that turns gas-inefficient smart contracts into gas-efficient ones while maintaining the program logic.

Overall, this work has *four* major contributions.

(1) We propose ten gas-inefficient programming patterns.

(2) We design and implement a novel tool, GasChecker which is based on SE to detect these patterns in the bytecode of smart contracts.

(3) We propose to scalalize the automated analysis by parallelizing SE according to the MapReduce programming model and designing a feedback-based load balancing strategy (FBLB) to improve resource utilization.

(4) We conduct a large-scale empirical study by applying GasChecker to the deployed smart contracts. Experiments show that GasChecker can find lots of gas-inefficient code in existing smart contracts with low false positives. Moreover, event recent compilers cannot eliminate all gas-inefficient patterns, and much money can be saved by optimizing the gas-inefficient code.

Paper organization. Section 2 introduces background knowledge with a motivating example. Section 3 describes the ten gas-inefficient patterns. Section 4 details the design of GasChecker. The scalability of GasChecker is evaluated in Section 5, and the results of the empirical study is presented in Section 6. The threats to validity are discussed in Section 7. After reviewing related studies in Section 8, we conclude the paper in Section 9.

2 Background and a Motivating Example

Blockchain. A *blockchain* is a continuously growing list of records, called *blocks* which contains zero or more transactions [11]. A *transaction* refers to a signed data package that contains a message sent by an account [12].

Ethereum. Ethereum is the largest blockchain with the capability of running smart contracts. Ethereum runs smart contracts in a stack-based virtual machine, so-called the Ethereum Virtual Machine (EVM). Ethereum has two kinds of accounts, external owned accounts (EOAs) and smart contracts. A smart contract contains executable bytecode, while an EOA does not.

Smart contract. A smart contract is typically written in a high-level language (e.g., Solidity) and then compiled into EVM bytecode. A smart contract is deployed by sending a special transaction whose *receiver* field is empty and its input data field carries the bytecode [12]. A smart contract is invoked by a transaction, where the *receiver* field gives the address of the callee and the *input data* field indicates the function to be invoked and the arguments.

Gas. To prevent resource abuse, especially DoS attacks [7], Ethereum proposes *gas* mechanism to charge execution fee from transaction senders. Execution fee is computed by *gas price* $\times$ *gas cost*, where gas cost is the amount of gas consumed measured by the number of units [6]. The gas cost of individual operation is defined by Ethereum's core protocol [12]. Some operations are cheap, e.g., ADD, AND, EQ, POP, because they are pure stack operations [6]. Some operations are expensive, e.g., SSTORE for updating the storage, CREATE for creating a smart contract [6].

Symbolic execution. Symbolic execution (SE) is a *multi-path* program analysis technique, which can exhibit different behaviors of the analyzed program corresponding to different paths. Moreover, by leveraging a theorem prover, SE can determine path feasibility, and therefore techniques based on SE can avoid analyzing dead code. Besides, SE maintains the symbolic expressions of variables, and therefore SE can reason about the relation of two expressions (e.g., whether two expressions are equal). The reasons for choosing SE rather than standard compiler techniques are explained in detail in Section 4.1.

MapReduce. MapReduce is a programming model and an associated implementation for processing and generating large data sets with a parallel, distributed algorithm on a cluster [13]. A MapReduce application consists of a Map procedure to perform filter and sort, a Reduce procedure to summarize, and a Driver procedure to integrate the Map and Reduce. MapReduce can launch multiple Map tasks (a Map task is an execution of the Map procedure) and run

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TETC.2020.2979019, IEEE Transactions on Emerging Topics in Computing


```
1  contract ExpensiveStorage {
2      uint num; //in storage
3      function setVar(uint x) {
4          for(uint i = 0; i < x; i++){
5              num++; //expensive
           }
       }
   }
```

(a) Gas-inefficient contract

```
1  contract CheapStorage {
2      uint num; //in storage
3      function setVar(uint x) {
4          uint y = num;
5          for(uint i = 0; i < x; i++){
6              y++; //cheap
           }
8          num = y;
       }
   }
```

(b) Efficent version

Fig. 1. A gas-inefficient contract and its efficient version

them independently and execute Reduce tasks (a Reduce task is an execution of the Reduce procedure) in parallel. MapReduce has several features. First, Reduce tasks will not be started until all Map tasks finish their jobs. Second, all Mappers (the workers run Map tasks) are independent, and hence there are no communications among Mappers. Hence, cloud services (e.g., Spark [14]) can simply re-execute a failed Map task on another worker for failure recovery. Third, a MapReduce application is expected to be scalable to the increase of workers since the communications among workers are greatly reduced by design.

A motivating example. Fig. 1(a) shows a smart contract containing a gas-inefficient pattern (i.e, *P3*, Section 3.2.1) and Fig. 1(b) gives its efficient version. It is worth noting that even a recent compiler (Solidity 0.5.10 released on Jun. 25, 2018) for smart contracts cannot perform this optimization. This contract declares a global variable *num*. Unlike traditional applications, global variables in a smart contract are stored in the storage. Since accessing the storage is expensive [6], reading and writing *num* in a loop repeatedly (Line 5) costs lots of gas. By contrast, the contract in Fig. 1(b) is more efficient with the same functionality as the costly contract. In particular, it reads *num* into a local variable y, then increases y in the loop, and finally writes y back to *num*. Hence, the number of access to storage reduces from $2x$ (1 read and 1 write in every iteration of the loop) to 2. Since the local variable y is stored in the stack and accessing the stack requires orders of magnitude less gas than operating storage [6], the code in Fig. 1(b) is gas-efficient.

It is worth noting that from the viewpoint of traditional program optimization, the code in Fig. 1(a) cannot be optimized for several reasons. First, accessing a local variable has no distinct difference with accessing a global variable in terms of speed. Besides, more memory is required to store the local variable when running the code in Fig. 1(b). Third, the code in Fig. 1(b) contains more instructions and hence more disk space is needed to store the code, more memory will be allocated to load the code, and more instructions will be executed. Hence, a traditional compiler would consider the code in Fig. 1(a) be better because it typically aims at improving runtime performance, reducing memory and disk consumptions. `GasChecker` can detect such gas-inefficient pattern in the bytecode of smart contracts as detailed in Section 4.6.

3 Gas-Inefficient Programming Patterns

We identify ten gas-inefficient programming patterns, and divide them into four categories.

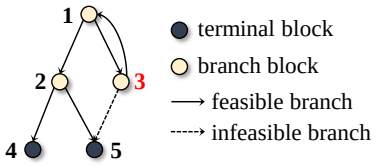

Fig. 2. An opaque predicate in block 3, but no dead code

3.1 Useless Code

3.1.1 P1: Opaque Predicate

An opaque predicate refers to a comparison that has only one outcome (i.e., *True* or *False*). Hence, the gas for invoking the smart contract can be reduced by removing the comparison, and the gas for deploying the smart contract can also be reduced by removing the code on the infeasible branch. It is non-trivial for the developers of smart contracts to avoid opaque predicates because opaque predicates do not alter program logic.

3.1.2 P2: Dead Code

The code that cannot be reached at runtime is dead. The gas for deploying smart contracts can be reduced by eliminating dead code. It is worth noting that opaque predicate is not always the same as dead code even if they often appear together (as shown in Fig. **??**). Dead code is the code that cannot be executed in practice, and an opaque predicate (also termed by invariant opaque predicate in recent studies [15]) is a comparison which always produces the same result. First, an opaque predicate does not necessarily result in dead code. Fig. 2 shows an example. Blocks 1, 2, 3 are branch blocks which end with a JUMPI, and blocks 4, 5 are terminal blocks which end with a STOP. Note that since STOP terminates the execution of smart contracts [6] they do not have successors. A solid edge indicates a feasible branch, while a dashed edge stands for an infeasible branch. Hence, there is an opaque predicate in block 3, because only one branch from block 3 is feasible. However, there is no dead code in this example, because block 5 can be reached from block 2. On the other hand, dead code is not necessarily caused by opaque predicates, e.g., functions never been called, the code after a return statement.

3.2 Loop

3.2.1 P3: Expensive Operations in a Loop

Expensive operations in a loop are worthy of attention because a single transaction can execute expensive operations multiple times, resulting in high money cost. A transaction sender sets the field *gas limit* of the transaction, indicating the amount of gas the sender can afford to execute that transaction. As a consequence, if the execution of a transaction requires more gas than the *gas limit*, the transaction will fail and the gas in the amount of *gas limit* will be consumed. By moving expensive operations outside the loop, the gas for invoking smart contracts can be reduced. However, the size of bytecode after optimization will be increased (as shown in Fig. 1), thus the gas for deploying smart contracts will be raised. It is worth noting that a contract can be deployed only once but invoked many times. Hence, such optimization can reduce overall gas consumption.

3.2.2 P4: Fusible Loops

Two loops are fusible, if they can be merged into one loop without changing program behaviors. By loop fusion, the gas for deploying smart contracts can be reduced because the size of bytecode becomes smaller, and the gas for invoking smart contracts can be reduced because the executed instructions will be reduced. Moreover, other optimizations (e.g., common subexpression elimination) can be applied on the merged loop body so that gas consumption can be reduced further. Although loop fusion has been applied in modern compilers, compilers may fail to find all fusible loops, particularly the loops whose loop bounds are hard to determine using standard compiler techniques.

3.2.3 P5: Repeated Computation in a Loop

This pattern refers to the computation that produces the same result in each iteration of a loop. By moving the computation outside the loop and using a temporary variable to hold the result, the optimized version saves the gas for executing smart contracts at the price of larger bytecode (i.e., more gas for deploying smart contracts). Since a contract is deployed once and executed many times, such optimization is acceptable.

3.2.4 P6: Unilateral Comparison in a Loop

This pattern means a comparison in a loop whose outcome is fixed. Please note that it is not an opaque predicate (*P2*) because the comparison can produce different results under different contexts. By moving the unilateral comparison outside the loop and copies the loop body, the optimized version saves gas for executing smart contracts since it reduces the number of comparisons at the price of larger bytecode.

3.3 Wasted Disk Space

3.3.1 P7: Redundant SSTORE

This pattern indicates *a storage that is never used (e.g., SLOAD) after definition (i.e., SSTORE)*. Please note that not only SLOAD but also some other operations (e.g., BALANCE obtains the balance of an account) read storage. Redundant SSTORE operations waste disk space and much money as the gas cost of SSTORE (as shown in Table 1) is high. Standard compiler techniques for detecting redundant variable definitions (e.g., liveness analysis) may not be suitable for our purpose because storage is like a database on the disk rather than a data structure in physical memory. In particular, a storage can be written by a transaction and then read by a subsequent transaction. Therefore, we need multi-path analysis and we cannot say that an SSTORE is redundant if the written location is not read in one program path. After optimization, both the gas for deploying and executing smart contracts can be reduced.

3.4 Gas-Inefficient Operation Sequence

This type of gas-inefficient patterns refer to consecutive EVM operation sequences that can be replaced with gas-efficient operation sequences. Although the following three patterns are specific to EVM bytecode, the basic idea can be extended to other smart contract platforms. After optimization, both the gas for deploying and invoking smart contracts can be reduced.

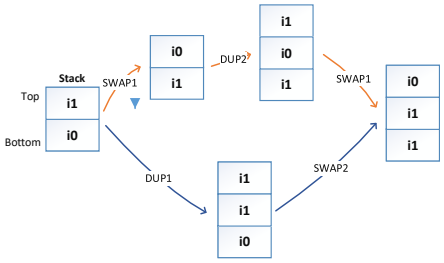

Fig. 3. Stack after the execution of SWAP1/DUP2/SWAP1

3.4.1 P8: SWAP1/DUP2/SWAP1

In EVM, SWAP1 exchanges the 1st and 2nd stack items and DUP2 duplicates the 2nd stack item [6]. Fig. 3 shows the change of stack after execution of the operation sequence SWAP1/DUP2/SWAP1 following the yellow edges, which consists of three operations. *i0* and *i1* indicate two stack items. This pattern can be simplified to DUP1/SWAP2 (in blue edges of Fig. 3) which is shorter and consumes fewer gases. SWAP2 exchanges the 1st and 3rd stack items [6]. Note that each swap and duplication needs 3 units of gas for execution [6], and hence the shorter sequence saves 3 units of gas while producing the same output. The opcode of each operation is a non-zero byte and each non-zero byte attached in a transaction needs 68 units of gas [6]. Besides, the bytecode of the gas-inefficient pattern is 0x908190 and the bytecode of the gas-efficient counterpart is 0x8091 [6]. Therefore, compared to the gas-efficient counterpart, the gas-inefficient pattern wastes 68 units of gas for contract deployment.

3.4.2 P9: PUSHx/POP

There are 32 push operations in EVM, including PUSH1, PUSH2, ..., PUSH32, which place 1 byte, 2 bytes, ..., 32 bytes on the stack, respectively [6]. Please note that push operations are the only EVM operations that have an operand. No matter how many bytes are placed, a push operation adds 1 item on the stack. Therefore, the operation sequence PUSHx/POP, $1 \leq x \leq 32$ makes no changes to stack, and hence such sequence can be removed to save gas. Five units of gas can be saved for contract execution by eliminating this pattern because a push operation and a pop operation cost three units and two units of gas, respectively [6]. The amount of gas saved for contract deployment by eliminating this pattern depends on the operand of the push operation and the upper bound is 2,312 (68 units for the pop operation, 68 units for the push opcode and 68×32 for the 32-byte operand without one zero byte) units of gas.

3.4.3 P10: PUSH1/NOT

NOT is a bitwise *not* operation that flips every bit of the 1st stack item [6]. PUSH1 pushes 1 byte on the stack, where the byte should be a constant [6]. Hence, the 1st stack item after the execution of NOT can be computed in advance. Consequently, the gas-inefficient pattern PUSH1/NOT can be replaced with a single PUSH1 whose operand is the bitwise *not* of the operand of PUSH1 in the gas-inefficient pattern. The efficient counterpart saves three units and 68 units of gas for contract executing and deploying, respectively, because the execution of one NOT needs three units of gas [6].

Table 1 summarizes the effects by replacing gas-inefficient patterns with their gas-efficient counterparts,

TABLE 1
Gas cost changes by replacing gas-inefficient patterns with their gas-efficient counterparts

gas change	*P1*	*P2*	*P3*	*P4*	*P5*	*P6*	*P7*	*P8*	*P9*	*P10*
execute	–	o	–	–	–	–	–	–	–	–
deploy	–	–	+	–	+	+	–	–	–	–

+:increase –: decrease o: no impact

where $+, -$ and O stand for increase, decrease and no impact, respectively.

4 GasChecker

4.1 Why Symbolic Execution?

GasChecker is based on SE, which is a heavy-weight program analysis technique. We adopt it for the following reasons. First, SE is a *multi-path* program analysis technique, which facilitates the detection of *P7* (Section 3.3.1). More precisely, to detect redundant SSTORE (*P7*), we need to check the usage of SSTORE in every program path. Second, by leveraging a theorem prover (Z3 used by GasChecker), SE *is able to determine whether two expressions are equivalent*, including the case where expressions are computed by bitwise/arithmetic operations or even depend on inputs. Such case is challenging to standard compiler techniques. Therefore, SE is more suitable than standard compiler techniques for detecting the gas-inefficient patterns, *P4* (Section 3.2.2) and *P5* (Section 3.2.3). To detect fusible loops (*P4*), we need to check whether the loop bounds of two adjacent loops are equal. SE is a rational choice to detect *P4* especially the loop bounds that are computed by bitwise/arithmetic operations or even depend on inputs. Similarly, SE can facilitate detecting *P5* because it is able to check whether the results of two computations in a loop are equal, even if the results are not constants.

Moreover, SE *is able to determine path feasibility*, which is also a challenge to standard compiler techniques. Hence, SE outperforms standard compiler techniques in detecting the gas-inefficient patterns, *P1* (Section 3.1.1), *P2* (Section 3.1.2) and *P6* (Section 3.2.4). Since SE attempts to explore all feasible paths, we consider any predicate with single outcome as an opaque predicate *P1*, and the uncovered code as dead code (*P2*) after the completion of path exploration. Similarly, by leveraging SE, we know the results of a comparison in a loop under different contexts, thus SE facilitates the detection of *P6*. Furthermore, our recent study discovers that SE *is capable of identifying some control flow transfers* which trouble standard compiler techniques [16]. Since control flow information is critical to program analysis, missing some control flow transfers may lead to inaccuracy in detecting gas-inefficient patterns. For example, if a control flow transfer from block *A* to block *B* is not discovered, block *B* may be misidentified as dead code. Besides, if the missed control flow transfer is the back edge of a loop, the loop structure may not be discovered and therefore the loop-related patterns (*P3*, *P4*, *P5*, *P6*) may be missed.

4.2 Overview

Fig. 4 shows the architecture of GasChecker, which consists of a *Master* to run the *Driver* procedure, multiple *Mapper*s to run Map tasks and multiple *Reducers* to run Reduce tasks. Both *Mapper*s and *Reducer*s are workers, and there is at least

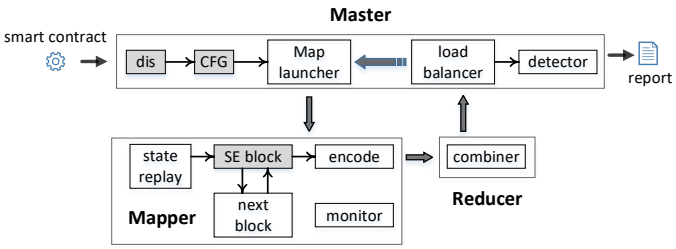

Fig. 4. Architecture of GasChecker

one worker. GasChecker takes in a smart contract and produces a report containing the locations of gas-inefficient patterns discovered in the analyzed smart contract. *Master* first disassembles the bytecode and constructs the control flow graph (CFG) from it. CFG is a directed graph, where each node denotes a basic block without control flow transfers inside and each edge indicates a control flow transfer (i,e., unconditional jump, conditional jump). Since it is difficult for standard compiler techniques to determine some jump targets, we complete the CFG during path exploration. After that, multiple Map tasks and Reduce tasks are launched. GasChecker runs Map tasks and Reduce tasks iteratively until all program paths have been explored or timeout. *Master* collects runtime information (e.g., value of an expression, executed basic blocks) from *Reducer*s for detecting gas-inefficient patterns, and performance information (e.g., time consumption for state recovery) for improving load balance. *Mapper*s accept Map tasks from *Master*, run CFG symbolically, and then produce new tasks if any. *Mapper*s recover the state, from which path exploration begins. New tasks are encoded in bit-vectors in order to improve the degree of parallelism by reducing network communications. The *monitor* in each *Mapper* records runtime information and performance information.

Most work is completed by *Mapper*s (e.g., path exploration) and the *Master* (e.g., gas-inefficient code detection), and the responsibility of Reduce tasks is to combine (i.e., reorganize) the outputs from Map tasks into a form that can be easily handled by the *Master* and then return the results to the *Master*. For example, if two Mappers output (*P1*, *info1*) and (*P2*, *info2*) where *P* and *info* stand for the encoded path prefix and information collected during SE, respectively, a *Reducer* combines them into ($\{P1, P2\}$, $\{info1, info2\}$). After receiving the combined results (e.g., $\{P1, P2\}$, $\{info1, info2\}$) from *Reducer*s, the *Master* generates new Map tasks from the first part $\{P1, P2\}$, then detects gas-inefficient patterns, and improves load balance using the second part $\{info1, info2\}$. We detail the key components of GasChecker in the following sections while omitting the details for implementing a standard SE engine. The components (in gray boxes) *dis* for dissembling bytecode, *CFG* for constructing the control flow graph and *SE block* for executing a basic block symbolically are reused from an existing sequential SE tool, OYENTE for detecting security bugs [17].

4.3 Task Encoding and State Recovery

Mapper receives a Map task from *Master*, explores the paths of the tested smart contract, forks new states when encounters conditional jumps (i.e., JUMPI), selects an unexplored state to execute, and outputs the unexplored states after the Map task finishes. The *state* in SE contains all necessary

information (e.g., program counter, symbolic expressions of all variables, constraints of the executed branches) that SE needs, and therefore SE can start path exploration from any states. In a decentralized environment, the Master should tell Mappers where (i.e., which basic block) to start path exploration and how much work should do (Section 4.5).

An intuitive way for *Master* to assign tasks to a Mapper is sending the exact state from which path exploration should start to that Mapper. After the *Mapper* finishes its work, all unexplored states are sent to *Reducer*s. Then, *Reducer*s aggregate the states from all *Mapper*s, and return the states to the *Master* for subsequent iterations of Map and Reduce routines. Although simple by design, this approach consumes considerable network resources, resulting in weak scalability when the number of workers increases.

To address this issue, we employ task encoding and state recovery for reducing the consumption of network resources. In particular, a Map task is represented by a bit-vector, indicating a path prefix. The state from which to start path exploration, can be recovered by symbolically executing the analyzed smart contract following the path prefix. For example, "0101" tells a Mapper to take *False*, *True*, *False*, *True* of the first four encountered branches when executing the analyzed smart contract, and then start path exploration after the fourth branch. Please note that state recovery neither check branch feasibility nor fork another state when encountering a branch, because the path prefix is feasible and the smart contract is executed along the path prefix during state recovery.

After the Mapper finishes, all unexplored states are encoded into path prefixes, indicating the paths from the beginning of the analyzed smart contract to the unexplored states. All path prefixes are feasible since their feasibility is validated by previous path exploration. The advantage of our method is that considerable network resources can be saved by sending bit-vectors rather than states. Therefore, compared to state transferring, our approach is more scalable at the expense of computing resources for state recovery. To increase the utilization of the computing resources of workers, we propose a feedback-based load balancing strategy, which is able to reduce the proportion of CPU resources for state recovery by dynamically adjusting the amount of work Mappers should do (Section 4.5).

4.4 Map Task

The Map procedure is the core component to implement SE. Beside the path prefix for state recovery and the CFG to explore, a Map task accepts a parameter *SN* from the Master, which stands for the number of EVM operations to be executed by the Map task. Hence, by changing *SN*, the Master can adjust the amount of work Mappers should do. After launching a *monitor* to collect runtime and performance information, the Mapper recovers the state from the given path prefix. After that, the Mapper begins path exploration until all paths have been explored or the designated amount of tasks, i.e., *SN* has been finished. A Mapper will not stop executing EVM operations until it executes a JUMPI or an operation (e.g., STOP) for terminating the execution of the smart contract, even if the number of executed operations exceeds the designated amount of tasks. The rationale lies in the fact that the Map task may not produce new Map tasks if SE terminates in the middle of a basic block. Operations are executed in sequence and a counter increases by 1.

When reaching a conditional jump (i.e., JUMPI), a new state will be generated if one branch can be taken and a new constraint will be added in the path condition, which is stored in the new state. The Mapper will select an unexplored state for subsequent SE. Any path exploration algorithms (e.g., depth-first search, random search) can be implemented in *Select_state()* to determine the execution priority of states. In the current implementation, `GasChecker` reuses the code of `OYENTE` for state prioritization, which is depth-first search. After the Mapper finishes its work, all unexplored states will be encoded into path prefixes. Finally, the Mapper outputs path prefixes together with the information collected during SE.

4.5 Feedback-based Load Balancing Strategy

MapReduce may result in low utilization of computing resources, because, *Mapper*s are independent and cannot communicate with each other, and therefore a Mapper cannot obtain new tasks which are generated by other *Mapper*s in the current iteration of MapReduce. That is, a *Mapper* has to wait until all *Mapper*s finish their jobs, and then the *Reducer*s process the outputs of *Mapper*s and return the combined data to *Master*. After that, *Master* launches new Map tasks. Consequently, the resource utilization would be low if some workers have to wait a long time for the other workers to finish their jobs.

To improve resource utilization, we propose a *feedback-based* load balancing strategy (FBLB), which adjusts the amount of work that should be completed by Mappers after each iteration. We first define two metrics to measure the performance of Mappers. *Replay ratio* stands for the proportion of time required for state recovery to the time for running Map tasks. Obviously, the lower the replay ratio is, the better. That is, more computing resources will be used for path exploration. Another metric is *working ratio*, which indicates the proportion of time for running tasks to the time the worker is available. Working ratio will be low if many workers wait for the completion of another worker. FBLB attempts to decrease the replay ratio and increase the working ratio. To this end, our approach limits the amount of work a Mapper should do to *SN*, which is an upper bound of the operations to be symbolically executed by a Mapper (does not count the operations executed during state recovery). Since there is a tradeoff between the two ratios, we cannot simply set *SN* with a fixed value. Section 5.2 will show that fixing *SN* is not an optimal solution.

Replay ratio will be lowered by increasing *SN* because a Mapper will use more time to execute more EVM operations with a larger *SN*. Working ratio can also be influenced by changing *SN*, however, the influence should be determined at runtime. For example, a large *SN* means that a Mapper needs a long time to finish its job, and hence the other workers may wait for it (i.e., working ratio of the current iteration may be low), especially in the case where workers outnumber tasks (i.e., there are workers that have no tasks to do at all). However, the larger *SN* is, the more new tasks may be generated by Mappers, and hence working ratio in subsequent iterations may be improved.

FBLB adjusts the amount of work as follows. First, GasChecker computes replay ratio (rr_i) and working ratio (wr_i) of iteration i from the performance date collected by the *monitor* of the Mapper. GasChecker increases SN_{i+1} from SN_i if rr_i is higher than a threshold *td*. Otherwise, GasChecker considers the adjustment of SN as a *search* problem and borrows the idea of simulated annealing algorithm [18] to approach the global optimal of working ratio. In particular, if $wr_i > wr_{i-1}$ (indicating the previous adjustment increases working ratio), we accept the previous adjustment and adjust SN_{i+1} based on SN_i (i.e., SN_{i+1} is a neighbor around SN_i). If $wr_i < wr_{i-1}$ (indicating that the previous adjustment decreases working ratio), we accept the adjustment with a probability defined as $e^{(wr_i - wr_{i-1})/T}$, where T is the temperature in simulated annealing. If the previous adjustment is rejected, SN_{i+1} will be set based on SN_{i-1}. Since the simulated annealing algorithm has a probability to accept a "bad" adjustment, our method could escape a local optimal.

4.6 Pattern Detection

P1. During SE, GasChecker records the result of each executed predicate. After SE, GasChecker regards a predicate as opaque, if the predicate has only one outcome.

P2. After CFG construction, GasChecker obtains all basic blocks of a smart contract. During SE, GasChecker records all executed blocks. After SE, GasChecker regards a basic block as dead, if it belongs to the smart contract but it is not executed by SE.

P3. To detect loop-related patterns (i.e., *P3* – *P6*), GasChecker first identifies loops through three steps. First, GasChecker scans the CFG to find back edges that indicate the existence of a loop, and identifies the entry block and exit block of a loop. We define the distance between two blocks as the least number of edges from one block to the other. Second, for each block, GasChecker calculates the distance between it and the entry block as well as the distance between it and the exit block. Finally, a block is regarded as being in a loop if it is closer to the exit block than to the entry block, because the block in a loop should go through the exit block to reach the entry block. Our method can also detect nested loops because it captures the structure of a loop in CFG. After identifying loops, P3 detection is simple. Technically, GasChecker scans each block in a loop for expensive operations. The current implementation of GasChecker can detect three expensive operations: SLOAD, SSTORE and BALANCE. The extension to support other operations is straightforward.

P4. GasChecker first identifies two adjacent loops and then computes their loop bounds. GasChecker considers two adjacent loops are fusible if their loop bounds are equal. If loop bounds are constants, it is easy to identify whether they are equal. If their loop bounds depend on symbols, we apply SE to determine whether they are equal. Our detection of fusible loops may introduce false positives because we do not consider data dependency (e.g., the second loop reads a variable which is set in the first loop) of loops. Manual analysis reveals that the false positive rate is low in practice (Section 6.2).

P5. GasChecker detects P5 in third steps. First, it detects loops by the approach described above. Second, for each computation in a loop, GasChecker records its outcome in each iteration of the loop during SE. Third, after SE, for each computation in a loop, GasChecker checks whether it results in the same outcome for every iteration. If so, GasChecker detects a repeated computation in a loop.

P6. GasChecker detects P6 in three steps. First, it detects loops by the approach described above. Second, it records the outcome of each branches inside loop during SE. Third, after SE, GasChecker detects a unilateral comparison, if the following two conditions are satisfied. First, the comparison produces two feasible branches, i.e., it is not an opaque predicate (*P2*). Second, the comparison produces only one outcome under the same context. Here, we define the *context* as the path condition before entering the loop.

P7. GasChecker detects P7 in two steps. First, during SE, GasChecker records the places of storage operations happened on each explored path, including SSTORE for writing and the operations for reading (e.g., SLOAD, BALANCE, EXTCODESIZE, EXTCODECOPY). Second, after SE, GasChecker regards an SSTORE as redundant, if the place written by the SSTORE can never be read in all paths. Note that we do not consider an SSTORE is redundant, if it sets a storage value to zero from non-zero, because such operation clears the storage [6]. Storage clearance reclaims disk space, so Ethereum refunds the transaction sender [6]. Therefore, such SSTORE does not waste gas.

P8 – P10. GasChecker detects P8 – P10 in two steps. First, GasChecker records each executed EVM operation during SE. Second, after the exploration of each program path, GasChecker checks whether the recorded EVM operations contain these patterns.

5 Scalability Evaluation

In this section, we evaluate the scalability of GasChecker by answering the following research questions.

RQ1: How is the scalability of GasChecker with the increase of workers?

RQ2: How is the effectiveness of FBLB?

5.1 Scalability of GasChecker

We use the top 10 most popular smart contracts (how to obtain smart contracts is described in Section 6.1), each of which is involved in at least 93,038 transactions, to evaluate the efficiency of GasChecker. By default, FBLB is turned on, and the threshold of replay ratio and initial *SN* (i.e., *SN* for the first iteration of MapReduce) are set as 30% and 200, respectively. To fairly compare the time consumptions required for analyzing contracts using different number of workers, we first record the number of paths explored for each analyzed contract when providing one worker. When testing with more workers, we stop analyzing a contract if the number of explored paths reaches the recorded number. We rent cloud resources from Google Cloud, and each node is equipped with 4 processors and 20GB main memory. Fig. 5 illustrates that the time needed for GasChecker to accomplish the analysis declines with the increase of workers.

We define *speedup* as the ratio of the time consumption given one worker to that given N workers, and define *speedup ratio* as the ratio of speedup to the number of

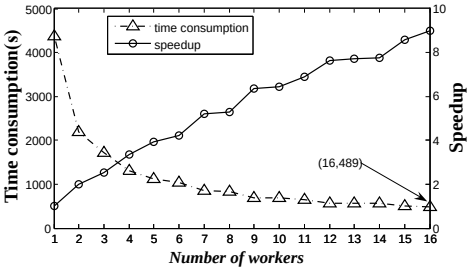

Fig. 5. Time consumption and speedup with various number of workers

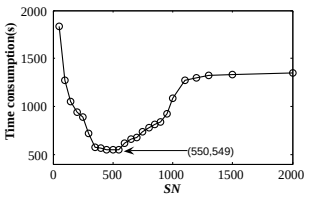

Fig. 6. Time consumption with different *SN*

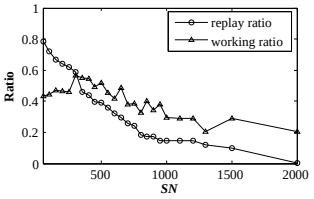

Fig. 7. Replay ratio and working ratio with different *SN*

workers. Fig. 5 shows that `GasChecker` is scalable because the *speedup* keeps increasing with the number of workers. In other words, we can expect *even better acceleration providing more workers*. For example, `GasChecker` with 16 workers need 489 seconds to complete analysis while `GasChecker` with 1 worker requires 4,377 seconds. In other words, by leveraging 16 workers, `GasChecker` obtains 9x speedup and the speedup ratio is about 56% (replay ratio is 15%, working ratio is 69%). It is worth noting that the *speedup ratio* of a practical distributed system cannot achieve 1 due to many practical issues such as the consumptions of communication, task dispatching, results collection. Moreover, `GasChecker` needs to consume CPU resources for state recovery, and there are some restrictions from MapReduce programming model (e.g., a Mapper cannot get new tasks directly from other Mappers), preventing full utilization of computing resources. We will show the effectiveness of FBLB in improving resources utilization in Section 5.2.

Answer to RQ1: `GasChecker` scales well with the increase of workers.

5.2 Effectiveness of FBLB

To evaluate the effectiveness of FBLB, we turn off it and launch `GasChecker` with fixed *SN*. Fig. 6 gives the execution time for analyzing the top 10 most popular smart contracts using different *SN*. We observe a downward trend and then an upward trend with the increase of *SN*. To find the reason for this observation, we draw the *replay ratio* in Fig. 7, and observe that *replay ratio* decreases with the increase of *SN*. Hence, `GasChecker` is inefficient with small *SN*, because considerable computing resources are used for state replay. For example, replay ratio is as high as 72% when *SN* is fixed as 100. Moreover, working ratio fluctuates with the increase of *SN*, and it becomes low given a large *SN* because some workers do lots of work, resulting in insufficient tasks for the other workers. For example, working ratio is as low as 20% when *SN* is fixed as 2,000. Another observation is that FBLB (489 seconds) is even better than the best result of fixed strategy (i.e., 549 seconds with a fixed *SN*, 550). The result is reasonable since our load balancing adjusts *SN* in every iteration of MapReduce to improve resource utilization while the fixed strategy uses the same *SN* in every iteration.

Answer to RQ2: FBLB is effective in improving resource utilization.

6 Results of the Empirical Study

This section reports the empirical study on the gas-inefficient patterns in the deployed smart contracts by answering the following questions.

RQ3: What is the analysis precision of `GasChecker`?
RQ4: How prevalent is gas-inefficient code in the deployed smart contracts?
RQ5: What is the difference of different compilers in terms of producing gas-inefficient code?
RQ6: How much money can be saved if the gas-inefficient code is optimized?

6.1 Data Set

We download all smart contracts (i.e., 599,934) that were deployed on the public blockchain of Ethereum from the launch of Ethereum, July 30th, 2015 to June 10th, 2017, and then choose the representative contracts from all contracts according to some criteria. We first describe how to obtain all smart contracts in two steps. The first step is to collect the addresses of all smart contracts. The second one is to get the bytecode of contracts by invoking an API that takes in the address of a contract. We detail the two steps as follows.

It is non-trivial to get the addresses of all smart contracts because the address space is 2^{20} (an address is 20 bytes), and we cannot determine whether an address is used by a contract only according to the address. We propose to collect the addresses that are specified by transactions and internal messages, because both a transaction and internal message can deploy a smart contract. To collect the addresses of smart contracts which are deployed by transactions, we instrument the function *ApplyTransaction()* in \core\state_processor.go of the Ethereum client, which is responsible for executing transactions. To collect the addresses of smart contracts which are deployed by internal messages, we instrument the handlers of all operations (i.e., CREATE, CALL, CALLCODE, DELEGATECALL, STATICCALL and SELFDESTRUCT) that are able to send internal messages [6]. For each address, we download the bytecode of the smart contract by invoking the API *GetCode()* provided by the client of Ethereum[3]. The API returns the bytecode, or empty if the address corresponding to an EOA or the corresponding smart contract has been removed from the blockchain. Eventually, the bytecode of 599,934 smart contracts is downloaded.

We select 1,500 representative smart contracts for analysis from all 599,934 contracts according to the following criteria. First, since we find that many contracts have identical bytecode, we only consider contracts with unique bytecode. Second, we include all open-source contracts. In particular, we search open-source smart contracts in Etherscan, and find 803 open-source contracts with unique bytecode. We also collect the source code of those 803 contracts from Etherscan for the comparison of different compilers (Section 6.4). Then, we choose 500 most popular smart contracts

3. Ethereum JavaScript API: https://github.com/ethereum/web3.js/

in terms of the number of transactions invoking smart contracts. To count such number, we instrument the function *ApplyTransaction()* because it executes transactions. Note that the numbers of transactions sent to the contracts having the same bytecode are accumulated to the unique contract bytecode. Finally, we randomly select other 197 contracts. Those 1,500 contracts are representative because they represent 114,903 contracts, which account for about 20% of all downloaded contracts. Moreover, the top 500 most popular contracts represent 99,385 contracts which are involved in 6,739,363 transactions. Note that among all 28,502,131 transactions, 9,139,590 out of them are sent to smart contracts. Therefore, the top 500 contracts are involved in about 74% of the transactions sent to smart contracts.

The empirical study is conducted on 17 nodes, consisting of 1 Master and 16 workers. By default, FBLB is turned on. We set the timeout for analyzing each smart contract as 10 minutes. Besides, we need to set a loop bound n to control how many times a loop can unfold; otherwise, `GasChecker` may achieve low code coverage, resulting in inaccurate detection results, because long program paths (especially due to loops) will be explored without setting a loop bound. Consequently, much code may remain unexecuted when timeout. However, setting a proper loop bound is not easy, given that we have limited time to analyze each contract. We propose to evaluate a loop bound by the number of basic blocks executed by `GasChecker` because `GasChecker` should cover as many basic blocks (without considering dead code) as possible to ensure high detection precision. Therefore, we run `GasChecker` by setting various loop bounds ranging from 1 to 10, and we find that `GasChecker` covers the most basic block when the loop bound is set to 4. More precisely, Table 2 demonstrates that there are 235,877 basic blocks of 1,500 contracts in total, and 7,532 dead blocks (as shown in Table 3) are detected. Therefore, $81.3\% = 235,877/(235,877 + 54,346)$ of basic blocks are covered when the loop bound is set to 4. Differently, the ratios of covered basic blocks when the loop bound is set to 1 and 10 are 75.1% and 77.5%, respectively. Hence, we set the loop bound to 4 by default. `GasChecker` allows to adjust parameters (e.g., timeout, loop bound).

6.2 Precision Analysis

We first enumerate the possible factors in theory which can impair the analysis precision of `GasChecker`, and then analyze the causes of false positives happened in practice.

We examine the false positives manually due to the lack of ground truth. More precisely, we randomly select and scrutinize 100 occurrences of each gas-inefficient pattern. Since there are only 15 instances and 99 instances of *P4* and *P7*, respectively (see Table 3), we investigate all of them. Consequently, we manually check 914 instances of gas-inefficient patterns that are discovered by `GasChecker`. We do not examine the false negatives because there lacks ground truth about the exact number of gas-inefficient patterns in smart contracts. Investigation shows that for *P3*, *P5* to *P7*, and *P8* to *P10*, all 699 pieces of gas-inefficient code are not false positives. For *P1*, *P2* and *P4*, the false positives are 5, 14, and 4, respectively. Hence, the false positive rate is 2.5% (23/914). We manually check the reasons for those 23 false positives, and reveal that all false positives of *P1* and *P2* are due to the reason that there are some feasible program paths which are not explored during path exploration. Moreover, we find that all false positives of *P4* are due to data dependency of loops that is not considered by `GasChecker`.

```
54  for (uint i = 0; i < strikes_.length; i++) {
55    if (numOptions < 20) {
56      uint optionID = numOptions++;
57      options[optionID] = strikes_[i];
      }
    }
```

Fig. 8. Misidentified opaque predicate in a loop at Line 55

```
198  while(i < 4)
199  {
200    while(j < 4 ...){
...      //some code
204  }
205  if(hits == 0) throw;
```

Fig. 9. Misidentified dead code at Line 205

```
274  for (i = 0; i < numAccounts; i++ ) {
275    pctx10 = partnerAccounts[i].pctx10;
276    maxAcctDist = totalFundsReceived * pctx10 / TENHUNDWEI;
277    if (partnerAccounts[i].credited < maxAcctDist) {
278      totalDistPctx10 += pctx10;
279    }
280  }
281  for (i = 0; i < numAccounts; i++ ) {
282    pctx10 = partnerAccounts[i].pctx10;
283    acctDist = distAmount * pctx10 / totalDistPctx10;
...    //some code
```

Fig. 10. Misidentified fusible loops at Lines 274, 281

Fig. 8 presents a misidentified opaque predicate (*P1*) in an open-source contract which was deployed at 0xaf0f6A53269Fc9DBbd9DA9F11c368d36B7A60006. The variable *numOptions* is initialized as 0, so the outcome of the comparison (Line 55) should be always *True* if the loop count is smaller than 20. On the contrary, the outcome of the comparison becomes *False* if the loop count is no smaller than 20. However, `GasChecker` misidentifies the comparison as an opaque predicate, because it unfolds the loop (Line 54) four times. Fig. 9 presents misidentified dead code (*P2*) in an open-source contract which was deployed at 0x7698392fff47d8d4CeE21295aD1F31b6ced9ad66. We keep the line numbers in the source file. `GasChecker` considers Line 205 is dead code, which is a false positive. Before Line 205, there is a nested loop that the basic blocks inside the inner loop can execute up to 16 times. However, `GasChecker` sets the loop bound as four, so the program path will be terminated before it reaches Line 205. Fig. 10 shows misidentified fusible loops (*P4*) in an open-source contract which was deployed at 0x61F9d1cE56aC1623eD4e949D7D420251fef0896. `GasChecker` considers the two loops at Line 274 and Line 281 are fusible, because they have the same loop bounds. Unfortunately, it is a false positive because the second loop reads a variable *totalDisPctx10* (Line 283) whose value is computed by the first loop (Line 278). Therefore, the second loop should be executed after the execution of the first loop due to such data dependency.

Then, we investigate whether we can eliminate the false positives of *P1* and *P2* by simply setting a larger loop bound. Therefore, we keep all default settings but set the loop bound to 50. Interestingly, all false positives are not removed and the analysis of the corresponding contracts triggers timeout. The result is accordant with the observation that `GasChecker` executes fewer basic blocks when the loop bound is set as ten than the number when the loop bound is set as four, given the same timeout setting (Section 6.1).

TABLE 2
Statistics of 1,500 smart contracts

# methods	Size (byte)	Time(s)	# blocks	# paths	# JUMP	# JUMPI	# checks
19,877	5,363,932	626,130	235,877	96,967	650,485	375,135	750,270/ 41,368

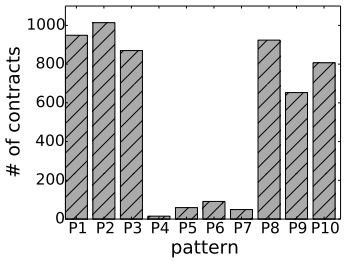

Fig. 11. Number of smart contracts containing gas-inefficient patterns

Fig. 12. Contracts with expensive operations in a loop

We then set the timeout for analyzing each contract as 300 minutes, which is 30 times longer than the default setting. In this setting (i.e., timeout = 300m, loop bound = 50), all false positives of *P1* and *P2* are eliminated, and the time consumed by analyzing the corresponding contracts (19 contracts) is 3,842 minutes.

Answer to Q3: `GasChecker` produces low false positive ratio (2.5%), which can be further reduced to 0.4% ($4/914$) by setting a larger loop bound and giving longer time for analysis.

6.3 Prevalence Analysis

Table 2 presents the statistics of the analyzed 1,500 smart contracts. Column 1 is the number of methods which can be invoked. Column 2 presents the overall size of those contracts in bytes. Hence, the average size of a smart contract is 3,576 bytes. The analysis is completed in 626,130 seconds (about 174 hours, column 3). Table 2 also gives the number of executed basic blocks (column 4), the number of explored paths (column 5), the number of unconditional jumps (column 6), the number of conditional jumps (column 7) and the number of queries to the theorem prover, `Z3`. The figure before "/" is the number of queries for identifying branch feasibility while the figure after "/" is the number of queries for solving the constraints for detecting *P4* and *P5* (e.g., query whether two loop bounds are equal). Table 3 presents the number of dead blocks (i.e., uncovered basic blocks), so the total number of basic blocks in 1,500 contracts are 290,223 ($235,877 + 54,346$), and thus there are 193 basic blocks per contract on average.

Fig. 11 shows the number of smart contracts containing each gas-inefficient patterns. Please note that a smart contract will be counted twice if it contains two different patterns. We observe that a large proportion of smart contracts contain *P1*, *P2* or *P3*. A reasonable explanation is that they could be easily introduced in writing smart contracts and compilers cannot eliminate all of them. We also notice that lots of contracts contain *P8*, *P9* and *P10*. A possible reason is that such low-level and architecture-dependent patterns are hard to be avoided by the developers of smart contracts. Moreover, the optimizations conducted by the compilers do not handle such low-level patterns. We evaluate the effectiveness of the latest compiler with optimization in reducing gas-inefficient code in Section 6.4. Fig. 12 illustrates the numbers of different expensive operations in a loop (*P3*). We find that SLOAD (803) and SSTORE (752) are more prevalent than BALANCE (246). Interestingly, if a loop contains an SSTORE (752), it is likely to contain a SLOAD together (744), but the opposite is not always true. Specifically, there are 90 = 81 + 9 contracts (i.e., more than 10% of total contracts containing at least one expensive operations) that have SLOAD but no SSTORE in a loop. The result matches the observation from Table 3 that redundant SSTORE occurs infrequently. Furthermore, 209 smart contracts (24%) contain three expensive operations together.

Fig. 13 shows the relationship between the size of a smart contract and the occurrences of gas-inefficient patterns in it. A point (x, y) in this figure indicates that there is a smart contract of x bytes and it has y occurrences of gas-inefficient patterns. As expected, *larger smart contracts are likely to contain more patterns*. Fig. 14 depicts the relationship between the number of transactions of a smart contract and the occurrences of gas-inefficient patterns in it. A point (x, y) in this figure denotes that there is a smart contract, which is involved in x transactions and has y occurrences of gas-inefficient patterns. Interestingly, *the more popular contracts (i.e., take part in more transactions) are less likely to contain many gas-inefficient patterns*. It may suggest that widely-used contracts are developed by professional software vendors or experienced programmers, and thus have better quality.

Table 3 gives the detailed results of this empirical study. The second row presents the total occurrences of ten gas-inefficient patterns in the 1,500 analyzed smart contracts. The third row gives the averages. For example, five (row 3, column 3) means that there are five opaque predicates on average for each contract. The remaining rows concern the 803 open-source smart contracts, which will be explained in Section 6.4.

Answer to Q4: A large proportion of smart contracts contains gas-inefficient patterns. Besides, opaque predicates, dead code, expensive operations in a loop and three low-level patterns occur frequently per contract.

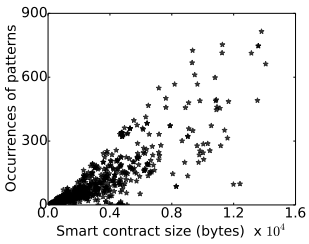

Fig. 13. Relationship between the occurrence of patterns with the size of bytecode

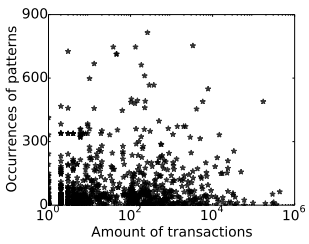

Fig. 14. Relationship between the occurrence of patterns with the # of involved transactions

6.4 Effectiveness of Compiler in Eliminating Gas-Inefficient Code

Developers can use different compilers (e.g., Solidity, Serpent, LLL) to convert the source code of smart contracts into EVM bytecode. Each compiler may have many different versions with different capabilities to optimize the bytecode. For example, Solidity has more than 350 versions whose first version is 0.1.1, released in 2015[4]. Developers can also turn

4. Remix: https://remix.ethereum.org/

TABLE 3
Number of occurrences of ten gas-inefficient patterns

		P1	*P2*	*P3*	*P4*	*P5*	*P6*	*P7*	*P8*	*P9*	*P10*
Total	sum	7,532	54,346	28,192	15	182	401	99	9,238	9,810	6,464
	ave	5	36.2	18.8	0.01	0.12	0.27	0.066	6.2	6.5	4.3
Open source (original)	sum	2,083	18,269	8,688	6	68	122	20	3,403	1,821	2,406
	ave	2.6	22.8	10.8	0.0075	0.085	0.15	0.025	4.2	2.3	3
Open source (0.4.0)	sum	2,274	11,706	9,161	5	13	171	14	1,040	1,174	1,291
	ave	2.8	14.6	11.4	0.0062	0.016	0.21	0.017	1.3	1.5	1.6
Open source (0.4.4)	sum	2,274	11,699	9,153	5	13	148	13	1,034	1,118	1,283
	ave	2.8	14.6	11.4	0.0062	0.016	0.18	0.016	1.3	1.4	1.6
Open source (0.4.8)	sum	2,274	11,629	9,075	5	12	145	10	1,030	0	1,275
	ave	2.8	14.5	11.3	0.0062	0.015	0.18	0.012	1.3	0	1.5
Open source (0.4.12)	sum	2,137	11,169	8,015	4	11	145	10	1,017	0	1,243
	ave	2.7	13.9	10	0.005	0.014	0.18	0.012	1.3	0	1.5
Open source (0.4.16)	sum	2,101	10,723	7,987	4	11	57	10	1,009	0	1,240
	ave	2.6	13.4	9.9	0.005	0.014	0.071	0.012	1.3	0	1.5
Open source (0.4.19)	sum	1,861	10,646	7,929	3	11	57	10	998	0	1,212
	ave	2.3	13.3	9.9	0.0037	0.014	0.071	0.012	1.2	0	1.5
Open source (0.4.21)	sum	1,620	10,568	7,870	3	10	54	9	990	0	986
	ave	2	13.2	9.8	0.0037	0.012	0.067	0.011	1.2	0	1.2
Open source (0.4.24)	sum	1,597	10,523	7,750	3	10	54	9	987	0	981
	ave	2	13.1	9.7	0.0037	0.012	0.067	0.011	1.2	0	1.2

on/off optimizations. In this section, we investigate whether different compilers can remove gas-inefficient code.

For this experiment, we collect the bytecode as well as the source code of 803 open-source smart contracts from Etherscan. We term the bytecode downloaded from Ethereum as *original* bytecode. Then, we use `GasChecker` to detect gas-inefficient code in the original bytecode. After that, we compile those contracts using eight different versions of Solidity with full optimizations. We name the produced bytecode as *optimized* bytecode. Then, we also employ `GasChecker` to discover gas-inefficient code in optimized bytecode. Finally, we compare the results of checking original bytecode and optimized bytecode. Note that we select Solidity, whose versions range from 0.4.0 (released on September 6, 2016) to 0.4.24 (released on May 16, 2018) because Solidity is the most popular language for developing Ethereum smart contracts [12], which undergoes hundreds of upgrading. We do not select Solidity whose versions are lower than 0.4.0 because we find that many contracts cannot be compiled successfully by earlier versions of Solidity due to the new features introduced in new versions.

Evaluation results are presented in row 4 to row 21 of Table 3. The row *sum* gives the number of gas-inefficient instances in 803 open-source smart contracts and the row *ave* presents the average number of gas-inefficient instance in each open-source contract. We have several observations from the experimental results. First, the number of each gas-inefficient instances in the contracts compiled by a newer version is no more than that of the contracts compiled by an older version. In other words, compilers have been improved to reduce gas-inefficient code. Interestingly, *P9* could be totally eliminated since Solidity 0.4.8. Second, many gas-inefficient code snippets still cannot be removed by recent compilers (e.g., 0.4.24). For example, Solidity 0.4.24 leaves 987 instances of *P8* which are comparable to the number (1,040) of gas-inefficient code instances compiled by Solidity 0.4.0, which was released 20 months before (Sep. 2016).

We find several potential reasons to explain why compilers cannot eliminate all gas-inefficient code. First, among all tasks in developing compilers (e.g., introducing new features, fixing bugs), reducing gas consumption may not be the primary task. Second, compilers are unlikely to incorporate heavy-weight techniques (e.g., symbolic execution) to optimize smart contracts because compilers need to produce bytecode quickly. Consequently, some gas-inefficient patterns discovered by `GasChecker` are not identified by the latest Solidity. For instance, compilers may feel difficult to determine path feasibility, and hence they may leave many gas-inefficient patterns (i.e., *P1*, *P2*, *P6*) in the produced bytecode. Besides, it is non-trivial for compilers to determine whether two expressions are equivalent if those expressions are computed by bitwise/arithmetic operations or depend on inputs, and therefore compilers may fail to detect many instances of *P4*, *P5*. Moreover, as suggested in Section 4.1, compilers may fail to identify some control flow transfers (e.g., the jump targets computed by bitwise/arithmetic operations), which hinders thorough program optimization.
Answer to Q5: The improvement of compilers can reduce some gas-inefficient code, but many pieces of gas-inefficient code were still left.

6.5 Money Saved Analysis

It is interesting to present how much money can be saved by optimizing gas-inefficient code since saving money is the ultimate goal of detecting and optimizing gas-inefficient code. Here, we optimize the smart contracts containing the last three gas-inefficient patterns (i.e., gas-inefficient operation sequence: *P8*, *P9*, *P10*) through the following steps, because this work focuses on the detection of gas-inefficient code. We leave the optimization for the other seven patterns in future work. First, based on the report produced by `GasChecker` which gives the locations of all detected gas-inefficient instances, we replace each gas-inefficient instance with its efficient counterpart (Section 3.4). During replacement, we record the new locations of the basic blocks after the replaced code. For instance, if there is a basic block locating at 0x40 in the bytecode and there is a piece of gas-inefficient code SWAP1/DUP2/SWAP1 (*P8*) before the basic block, the new location of the basic block should be 0x3f because the gas-efficient counterpart is DUP1/SWAP2 which is one byte shorter than the gas-inefficient code.

After that, we modify the jump targets of JUMP and JUMPI which jump to the basic blocks whose locations are adjusted. Note that a smart contract will trigger an exception if a jump operation jumps to an incorrect location [6]. We use a semi-automated way to modify jump targets. In particular, we automatically search the bytecode of the code patterns PUSHx/JUMP and PUSHx/JUMPI whose jump targets need to be adjusted, and then change the operand of PUSHx to the new locations. Note that in the patterns PUSHx/JUMP and PUSHx/JUMPI, the targets of the jump operations are given as the operands of the push operations [6]. For the jump operations whose targets are obtained by the other fashions (e.g., the push operation locates in a different place, the target is computed by bitwise/arithmetic operations), we manually modify EVM operations to ensure the correctness of jump targets.

To evaluate the money saved in deploying the optimized smart contracts, we deploy the original contracts and the optimized contracts in our private Ethereum blockchain, which is isolated from the public Ethereum blockchain, for

the purpose of experiments. The amount of gas consumed for contract deployment is returned by the transaction for deploying that contract. By comparing the gas consumption for deploying the original contracts with that for deploying the optimized ones, we find that 3,991,702 units of gas are saved. Considering that *gas_price* is about 5×10^{-8} Ether and 1 Ether can be exchanged into around 200 USD in September, 2019, the saved money in deploying the optimized smart contracts is about 40 USD.

To evaluate the money saved in invoking the optimized smart contracts, we first instrument the EVM and record the execution trace for each execution of smart contracts during synchronization with the public Ethereum blockchain. An *execution trace* consists of all executed EVM operations of a smart contract triggered by one transaction. Then, we calculate the gas consumption for invoking the original smart contracts by accumulating the gas cost for each executed EVM operation in execution traces. After that, we replace the gas-inefficient instances in execution traces with the gas-efficient counterparts. Then, we compute the gas consumption for invoking the optimized smart contracts by accumulating the gas cost for each EVM operations in the adjusted execution traces. By comparing the gas consumption for invoking the original smart contracts with that for invoking the optimized contracts, we find that 147,908,132 units of gas are saved. Thus, the invoking of the optimized contracts saves about 1,480 USD. Considering that there are more than 8 million smart contracts and more than 500 million transactions in total until September 2019, and this experiment just optimizes 1,500 smart contracts which are involved in about 7 million transactions, the amount of money that can be saved by optimizing all 8 million smart contracts should be several orders of magnitude than the amount by optimizing 1,500 smart contracts.

Answer to Q6: About 1,520 USD can be saved by optimizing three gas-inefficient patterns in 1,500 smart contracts. We reasonably expect *significantly more* money can be saved if we optimize all deployed smart contracts to eliminate all ten gas-inefficient patterns.

6.6 Case Studies

This section conducts case studies on 4 deployed smart contracts covering 4 gas-inefficient patterns. For the ease of presentation, we show open-source smart contracts and keep their line numbers unchanged. We neither show the practical cases of the first 3 patterns because they have been presented in our previous position paper, nor show last 3 patterns because they are low-level which have been explained in Section 3.4.

6.6.1 Big: P4

Big is deployed at 0xb36ce92cad11e7a9b903531f30590ebc2e99 1ea6, which has two fusible loops at Line 97 and Line 101, Fig. 15 (Top) respectively. By asking a theorem prover, `GasChecker` can find that their loop bounds (i.e., *category.VotesCount*) are equal, though they are not constant. though they are not constant. The first loop (Line 97) resets *totalVotes* to zero, and the second loop (Line 101) checks whether *totalVotes* is zero. Hence, the two loops can be combined and the comparison at Line 102 can be removed, as shown in Fig. 15 (Bottom).

```
97   for(i = 0; i < category.VotesCount; i++){
98     temporary[category.Votes[i].From].totalVotes = 0;
99   }
100
101  for(i = 0; i < category.VotesCount; i++){
102    if(temporary[category.Votes[i].From].totalVotes == 0){
103      temporary[category.Votes[i].From].rank =
             category.Ranks[category.Votes[i].From];
104    }
```

⇩

```
97   for(i = 0; i < category.VotesCount; i++){
98     temporary[category.Votes[i].From].totalVotes = 0;
99     temporary[category.Votes[i].From].rank =
         category.Ranks[category.Votes[i].From];
```

Fig. 15. Top: two fusible loops at Line 97 and 101. Bottom: efficient version

```
14   uint private Payout_id = 0;
15   uint private number_of_players = 0;
 ...
108  function CancelRoundAndRefundAll(){
109    if(number_of_players == 0) return;
110
111    if(last_time + time_max < block.timestamp){
112      for(uint i = Payout_id; i < (Payout_id + number_of_players); i++){
```

⇩

```
       ...
111    if(last_time + time_max < block.timestamp){
112      uint tmp = Payout_id + number_of_players;
113      for(uint i = Payout_id; i < tmp; i++){
```

Fig. 16. Top: a repeated computation in a loop at Line 112. Bottom: efficient version

6.6.2 Honestgamble: P5

Honestgamble is deployed at 0x7c4a690585e89c01aebfce188b 9bec8def9e8d, containing a repeated computation *Payout_id + number_of_players* in a loop at Line 112, Fig. 16 (Top). Since the two variables do not change in the loop, the computation result is the same in each iteration. As the two variables are in storage, the computation involves two expensive SLOAD operations to read them into the stack. It can be optimized by moving the comparison outside the loop, as shown in Fig. 16 (Bottom), so that the execution number of SLOAD is reduced from $2 \times$ *number_of_players* to 2.

6.6.3 PassFunding: P6

PassFunding is deployed at 0x055a9c349cDC2A598439d6A45 D0A83CAd3864FDc, containing a comparison with unilateral outcome (Line 1237) in a loop (Line 1225), as shown in Fig. 17 (Top). Since the variable *tokenCreation* is in the storage and changeable by other functions, its value cannot be determined in compilation. However, *tokenCreation* is never changed in the loop at Line 1225, and hence the comparison at Line 1237 produces the same result (i.e., *True* or *False*) in each iteration. The efficient code given in Fig. 17 (Bottom) moves the comparison outside the loop and copies the loop body into the *else* block. This optimization reduces the number of comparison from $\_to - \_from + 1$ to just 1.

6.6.4 Oraclize: P7

Oraclize is deployed at 0xf05782932dBABDe1D657a5311FC3b 78db81c60E7 (Fig. 18) that has two functions with the same name *setBasePrice()*, each of which has a redundant SSTORE (Line 110 and 116), because the private variable *baseprice* is never read on every feasible path in the smart contract. Therefore, the efficient code removes the declaration and storage operations of *basePrice* to save gas.

```
770   bool tokenCreation;
      ...
1225  for(uint i = _from; i <=_to; i++){
      ...
1237        if(tokenCreation)
1238            contractorManager.rewardToken(_partner, _amountToReward
                    partners[i].presaleDate);
```

```
770   bool tokenCreation;
      ...
1225  if(tokenCreation){
1226     for(uint i = _from; i <=_to; i++){
            ...
1238            contractorManager.rewardToken(_partner, _amountToReward
                    partners[i].presaleDate);
1239  else{
1240     for(uint i = _from; i <= _to; i++){
            ...
```

Fig. 17. Top: a unilateral comparison at Line 1237 in a loop. Bottom: efficient version

```
55    uint baseprice;
      ...
108   function setBasePrice(uint new_baseprice){
109      if((msg.sender != owner) && (msg.sender != cbAddress)) throw;
110      baseprice = new_baseprice;
111      for(uint i=0; i < dsources.length; i++) price[dsources[i]] = new_baseprice *
                price_multiplier[dsources[i]];
112   }
113
114   function setBasePrice(uint new_baseprice, bytes proofID){
115      if((msg.sender != owner) && (msg.sender != cbAddress)) throw;
116      baseprice = new_baseprice;
117      for (uint i=0; i < dsources.length; i++) price[dsources[i]] = new_baseprice *
                price_multiplier[dsources[i]];
118   }
```

Fig. 18. Two redundant SSTORE at Line 110 and 116

7 Threats to Validity

The first threat to the validity of our study results is the inherent limitations of SE. For example, to prevent path explosion, `GasChecker` unfolds the loops up to four times. The second threat is the imprecision of the detection module that will result in false positives, e.g., we do not consider data dependency when detecting fusible loops. We will improve the accuracy of `GasChecker` by leveraging advanced analysis techniques (e.g., data dependency analysis) and advanced techniques to process loops (e.g., loop summary) in future work. The third threat is the representativeness of the discovered gas-inefficient instances for the analysis of false positives. To minimize the threat, we randomly select 100 gas-inefficient instances for each pattern. Another threat is the representativeness of the analyzed smart contracts to evaluate the efficiency of `GasChecker`. To reduce the threat, we use the top 10 most popular contracts for evaluation. Moreover, when evaluating `GasChecker`, its parameters (e.g., default *SN*, the threshold of replay ratio) are set empirically. We will try other combinations of parameters and investigate how to set the parameters automatically in future work. The number of workers is also a threat, because we evaluate `GasChecker` on at most 16 workers. To reduce the threat, we evaluate the scalability of `GasChecker`, and find that `GasChecker` is scalable to the increase of workers. Hence, we can reasonably expect a good performance of `GasChecker` when it is deployed on more workers due to its scalability. The last threat is the potential errors in our manual analysis of false positives. To reduce the threat, we cross-validated the results by asking three Ph.d. students to check the reports independently.

8 Related Work

8.1 Gas-Related Studies

Chen et al. propose to defend against DoS attacks of Ethereum by adjusting gas costs dynamically [7]. By doing so, DoS attacks exploiting under-priced EVM operations will be terminated quickly [7]. `MadMax` decompiles EVM bytecode, and then detects gas-related security problems from the decompiled code [19]. `Gastap` derives the sound gas upper bounds for all public functions of a given smart contract, by inferring size relations, generating gas equations and solving the equations [20]. `Gasol` extends `Gastap` by replacing multiple accesses to the same storage location with one access [21]. GasFuzz applies feedback-directed fuzz testing to automatically generate inputs which could lead to a high gas consumption of contract functions [22]. Marescotti et al. leverages symbolic model checking to compute the exact worst-case gas consumption for smart contracts [23].

Yang et al. conduct an empirical study of gas consumption, and they have several observations, including some under-priced EVM operations that can be exploited by DoS attacks [24]. `GRuB` dynamically stores data in smart contracts or offline to reduce gas cost of data-intensive smart contracts [25]. Zhang et al. propose a novel data structure, so-called GEM^2-Tree to substitute the original Merkle hash tree in Ethereum to reduce gas cost [26]. `SmartCheck` [27] detects 21 kinds of code issues in Ethereum smart contracts. Two of them are related to gas-inefficient code. The first is *byte*[] because it can be replaced with *bytes* which is cheaper. The second is the loops with function calls inside because repeated function invocations will result in considerable gas consumption. `GasChecker` is different with `SmartCheck` mainly because `SmartCheck` relies on the source code of smart contracts. Unfortunately, open-source contracts only account for less than 1% [8]. For example, since there is no type information in EVM bytecode, it is difficult to distinguish *byte*[] from *bytes* without the source code.

8.2 Parallel Symbolic Execution

Staats et al. propose static task partition to divide the whole execution tree into disjoint subtrees and explore the subtrees in parallel [28]. Their method cannot balance the workload because the whole execution tree is unknown before execution [28]. Other parallel SE approaches [29], [30], [31], [32], [33], [34] partition the task dynamically, and move workload from busy nodes to idle nodes to improve resource utilization. However, Ibing's work [29] cannot always achieve good balancing due to the high coordination cost, low parallelism of tested units, and the operation delay of shared data-structures etc. `Cloud9` [30] balances well because an idle worker can directly acquire jobs from other busy workers, so the resources of workers can be well utilized. `SCORE` [32], `LCT` [33], and `ParSym` [34] are similar to `Cloud9` in task partition and workload migration. The major difference between those parallel SE tools [29], [30], [32], [33], [34] and `GasChecker` lies in that they are not tailored to the MapReduce framework. Hence, those

approaches cannot enjoy the benefits of commodity cloud services (e.g., scalability, flexibility, and fault tolerance).

9 Conclusion

We identify and summarize ten gas-inefficient patterns, and then propose and develop the system named `GasChecker` to detect gas-inefficient code in the bytecode of smart contracts. `GasChecker` is scalable to millions of smart contracts by parallelizing SE through MapReduce. Evaluation shows that it (1) scales well with the increase of workers; (2) FBLB is effective in improving resource utilization; (3) produces only a few false positives.

Acknowledgement

Ting Chen is partially supported by National Natural Science Foundation of China (61872057) and National Key R&D Program of China (2018YFB0804100). Xiapu Luo is partially supported by Hong Kong RGC Project (No. 152193/19E).

References

[1] M. von Haller Gronbaek. (2016) Blockchain 2.0, smart contracts and challenges. [Online]. Available: https://www.twobirds.com/en/news/articles/2016/uk/blockchain-2-0--smart-contracts-and-challenges
[2] J. Kehrli. (2016) Blockchain 2.0 - from bitcoin transactions to smart contract applications. [Online]. Available: https://www.niceideas.ch/roller2/badtrash/entry/blockchain-2-0-from-bitcoin
[3] A. Ruth. (2016) Why build decentralized applications: understanding dapp. [Online]. Available: https://due.com/blog/why-build-decentralized-applications-understanding-dapp/
[4] X. Li, J. Peng, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Generation Computer Systems*, 2017.
[5] S. Kalla. (2017) To the moon? blockchain's hiring crunch could last years. [Online]. Available: https://www.coindesk.com/moon-blockchains-big-hiring-crunch-last-years/
[6] G. Wood. (2017) Ethereum: A secure decentralised generalised transaction ledger. [Online]. Available: https://bravenewcoin.com/assets/Whitepapers/Ethereum-A-Secure-Decentralised-Generalised-Transaction-Ledger-Yellow-Paper.pdf
[7] T. Chen, X. Li, Y. Wang, J. Chen, Z. Li, X. Luo, M. H. Au, and X. Zhang, "An adaptive gas cost mechanism for ethereum to defend against under-priced dos attacks," in *ISPEC*, 2017.
[8] M. Fröwis and R. Böhme, "In code we trust? measuring the control flow immutability of all smart contracts deployed on ethereum," in *DPM&CBT*, 2017.
[9] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *SANER*, 2017.
[10] T. Chen, Y. Feng, X. Luo, X. Lin, and X. Zhang, "Cloud-based parallel concolic execution," in *SANER*, 2017.
[11] (2017) Blockchain. [Online]. Available: https://en.wikipedia.org/wiki/Blockchain
[12] (2017) Ethereum homestead documentation. [Online]. Available: http://www.ethdocs.org/en/latest/
[13] J. Dean and S. Ghemawa, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[14] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *HotCloud*. USENIX, 2010.
[15] J. Ming, D. Xu, L. Wang, and D. Wu, "Loop: Logic-oriented opaque predicate detection in obfuscated binary code," in *CCS*, 2015.
[16] T. Chen, Z. Li, Y. Zhang, X. Luo, T. Wang, T. Hu, X. Xiao, D. Wang, J. Huang, and X. Zhang, "A large-scale empirical study on control flow identification of smart contracts," in *Proc. ESEM*, 2019.
[17] L. Luu, D. H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *CCS*, 2016.
[18] (2017) Simulated annealing. [Online]. Available: https://en.wikipedia.org/wiki/Simulated_annealing
[19] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," in *OOPSLA*, 2018.
[20] E. Albert, P. Gordillo, A. Rubio, and I. Sergey, "Running on fumes," in *VECoS*, 2019.
[21] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Gasol: Gas analysis and optimization for ethereum smart contracts," in *TACAS*, 2020.
[22] F. Ma, Y. Fu, M. Ren, W. Sun, Z. Liu, Y. Jiang, J. Sun, and J. Sun. (2019) Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability. [Online]. Available: https://arxiv.org/pdf/1910.02945.pdf
[23] M. Marescotti, M. Blicha, A. E. Hyvärinen, S. Asadi, and N. Sharygina, "Computing exact worst-case gas consumption for smart contracts," in *ISoLA*, 2018.
[24] R. Yang, T. Murray, P. Rimba, and U. Parampalli, "Empirically analyzing ethereum's gas mechanism," in *EuroS&PW*, 2019.
[25] K. Li, Y. Tang, Q. Zhang, C. Xu, and J. Xu. (2019) Grub: Gas-efficient blockchain storage via workload-adaptive data replication. [Online]. Available: https://arxiv.org/pdf/1911.04078v1.pdf
[26] C. Zhang, C. Xu, J. Xu, Y. Tang, and B. Choi, "Gem^2-tree: A gas-efficient structure for authenticated range queries in blockchain," in *ICDE*, 2019.
[27] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *WETSEB*, 2018.
[28] M. Staats and C. Păsăreanu, "Parallel symbolic execution for structural test generation," in *ISSTA*, 2010.
[29] A. Ibing, "Parallel SMT-constrained symbolic execution for eclipse cdt/codan," in *ICTSS*, 2013.
[30] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *EuroSys*, 2011.
[31] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: a software testing service," vol. 43, no. 4, 2010.
[32] M. Kim, Y. Kim, and G. Rothermel, "A scalable distributed concolic testing approach: An empirical evaluation," in *ICST*, 2012.
[33] K. Kähkänen, T. Launiainen, O. Saarikivi, J. Kauttio, K. Heljanko, and Niemelä, "LCT: An open source concolic testing tool for java programs," in *BYTECODE*, 2011.
[34] J. Siddiqui and S. Khurshid, "Parsym: Parallel symbolic execution," in *ICSTE*, 2010.

Ting Chen received his PhD degree from University of Electronic Science and Technology of China (UESTC), China, 2013. Now he is an Associate Professor in UESTC. His research interest focuses on blockchain, smart contract and program analysis.

Youzheng Feng received his MS degree from UESTC. Now he serves in Alibaba Group.

Zihao Li received his Bachelor degree from UESTC. Now he is a master student in UESTC.

Hao Zhou is a PhD student in Hong Kong Polytechnic University.

Xiapu Luo received his PhD degree from Hong Kong Polytechnic University. After that, he continued his research as a Postdoctoral researcher in Georgia Institute of Technology. Now he is an Associate Professor in Hong Kong Polytechnic University. His research interest includes network measurement, mobile security and blockchain.

Xiaoqi Li is a PhD student in Hong Kong Polytechnic University.

Xiaoxiu Zhuo received his Bachelor degree from UESTC. Now he is a master student in UESTC.

Jiachi Chen received the B.S. degree in Institute of Service Engineering, HangZhou Normal University, China in 2016. He obtained M.S degree in Department of Computing, Hong Kong Polytechnic University, in 2017. After that, Jiachi spent one year at the Hong Kong Polytechnic University as a research assistant advised by Dr. Xiapu Luo. Currently, he is a Ph.D student in Faculty of Information Technology, Monash University, Australia.

Xiaosong Zhang received his PhD degree from UESTC, 2011, and now he is a full professor in UESTC. His research focuses on network security, AI security and blockchain security.