Large-Scale Empirical Study of Inline Assembly on 7.6 Million Ethereum Smart Contracts

Zhou Liao, Shuwei Song, Hang Zhu, Xiapu Luo, Zheyuan He, Renkai Jiang, Ting Chen, Jiachi Chen, Tao Zhang and Xiaosong Zhang

Abstract—Being the most popular programming language for developing Ethereum smart contracts, Solidity allows using inline assembly to gain fine-grained control. Although many empirical studies on smart contracts have been conducted, to the best of our knowledge, none has examined inline assembly in smart contracts. To fill the gap, in this paper, we conduct the first large-scale empirical study of inline assembly on more than 7.6 million open-source Ethereum smart contracts from three aspects, namely, source code, bytecode, and transactions after designing new approaches to tackle several technical challenges. Through a thorough quantitative and qualitative analysis of the collected data, we obtain many new observations and insights. Moreover, by conducting a questionnaire survey on using inline assembly in smart contracts, we draw new insights from the valuable feedback. This work sheds light on the development of smart contracts as well as the evolution of Solidity and its compilers.

Index Terms—Ethereum, smart contract, Solidity, inline assembly, Yul.

1 Introduction

Smart contracts, a kind of autonomous programs running on the blockchain, have been used to implement various applications due to the advantages inherited from the blockchain [1]. Ethereum hosts the largest number (more than 27 million [2]) of smart contracts, which are typically written in a high-level language (e.g., Solidity [3]) and then compiled into EVM (Ethereum Virtual Machine) bytecode. Unfortunately, such high-level languages do not support all functionalities desired by developers. To address this issue, Solidity, the most popular high-level language for developing Ethereum smart contracts [3], allows developers to embed inline assembly into Solidity source code in order to implement such functionalities [4].

Developing smart contracts with inline assembly has attracted great attention. Firstly, inline assembly has been widely used in smart contracts. For example, we investigate the use of inline assembly in Decentralized Finance (DeFi) [5] applications, which are currently very popular and important smart contract based applications in Ethereum. Defillama [6], a well-known DeFi data platform, provides a ranking list of the most funded DeFi (showing the top 295). We crawl the addresses of DeFi smart contracts from this platform and find that 233 of them are open-source, and 170 (73%) of these open-source DeFi use inline assembly, which is a very high number / proportion. Secondly, developers often encounter various problems in how to use inline assembly in smart contracts. By crawling all the posts about inline assembly for Ethereum smart contracts from two very popular developer forums, Stack Exchange [7] and Reddit [8], we found 718 questions raised by 478 users and 1,346 answers, where 676 questions are from StackExchange and 42 questions are from Reddit. 463 questions concern how to solve the problems encountered when using inline assembly in smart contracts. Besides, the answers to some other questions also recommend using inline assembly to solve their problems. By extracting the topics from these questions and answers relevant to inline assembly through the Latent Dirichlet Allocation (LDA) model [9], we choose five as the number of topics. More precisely, we select different numbers of topics to create the LDA models and evaluate the effect of each model according to the perplexity score [10]. Finally, we find that the best result is obtained when the number of topics selected is five, and these five topics include *memory, array, return, storage and event*. In detail, the first topic is related to memory operations in inline assembly, accounting for 48.1%; the second topic is related to using arrays in inline assembly, accounting for 23.5%; the third topic is returning value in inline assembly, accounting for 14.8%; the fourth topic is storage operations in inline assembly, accounting for 8.4%; the fifth topic is event use in inline assembly, accounting for 5.2%.

Although there are many empirical studies and surveys on smart contracts [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], [25], to the best of our knowledge, none has examined inline assembly in smart contracts. In addition, a few works mentioned inline assembly [26], [27], [28], [29], [30], [31], but they did not conduct in-depth research on inline assembly for design, security or other reasons (explained in more detail in §8). Although the usage of inline assembly in C has been investigated [32], its research methodology and the observations cannot be directly applied to the inline assembly of smart contracts, because

- *Z. Liao, S. Song, H. Zhu, Z. He, R. Jiang, T. Chen and X. Zhang are with the Institute for Cyber Security, University of Electronic Science and Technology of China, Chengdu 611731, China. (Corresponding Authors: Xiapu Luo, Ting Chen)*
  E-mail: brokendragon@uestc.edu.cn
- *X. Luo is with the Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Hong Kong.*
  E-mail: csxluo@comp.polyu.edu.hk
- *J. Chen is with the Monash University, Clayton VIC 3800, Australia.*
  E-mail: Jiachi.Chen@monash.edu
- *T. Zhang is with the Macau University of Science and Technology, Macao.*
  E-mail: tazhang@must.edu.mo

smart contracts and C programs have significant differences in instruction sets, running platforms, syntax and semantics of source code and inline assembly (explained in more detail in §8).

In this paper, we conduct the *first* large-scale empirical study of inline assembly on more than 7.6 million open-source Ethereum smart contracts to answer three research questions, **RQ1:** How is the prevalence of inline assembly in smart contracts? **RQ2:** What are the differences between the smart contracts with/without inline assembly in terms of five metrics detailed in §3.3. and **RQ3:** Why do smart contracts use inline assembly? The answers to these questions are very useful to the development of smart contracts, the evolution of Solidity, and the optimization of Solidity compilers. For instance, smart contract developers may be interested in how to use inline assembly to implement a functionality that cannot be realized by Solidity. A recent questionnaire survey shows that the lack of support for memory management is a limitation of Solidity and developers have to pay much effort to reduce gas (gas is explained in §2) consumption [33]. We show in §5 that inline assembly can overcome such limitation and save gas. Moreover, Solidity developers can benefit from the answers to the aforementioned questions to decide whether a new functionality should be supported by built-in features of Solidity or inline assembly and learn from inline assembly examples.

To answer the above research questions, we inspect and compare the smart contracts with/without inline assembly from three aspects, including source code, bytecode, and transactions. However, it is challenging to identify and extract the bytecode segment that is compiled from inline assembly from the entire bytecode of a smart contract for four reasons. First, there are no marks in EVM bytecode to distinguish such bytecode segment from the remaining bytecode compiled from Solidity. Second, the language for writing the inline assembly of smart contracts is Yul, which is an intermediate language that can be compiled into EVM bytecode [34]. Yul has high-level constructs (e.g., switch...case, if, for loops) [34], which cannot be directly translated into EVM instructions. Third, the optimization routines of Yul [34] make the generated bytecode very different from the Yul code. Finally, if the statements in an inline assembly fragment depend on the Solidity code, they cannot be compiled separately from the Solidity code. To address this challenge, we propose a *novel* approach that leverages crafted inline assembly statements to determine the start and the end of such bytecode segment and then extract it (details in §3.2).

After analyzing more than 7.6 million open-source Ethereum smart contracts, we obtain the answers to three RQs as well as new observations and insights. Note that the answers, new observations and insights in this paper are derived from the statistics of Ethereum open-source smart contracts, instead of all smart contracts. Investigating inline assembly in smart contracts without source code would be our future work because how to identify the bytecode generated from inline assembly statements without the help of source code is still an unsolved challenge.

Our answer to **RQ1** shows that inline assembly is prevalent in smart contracts because 11.9% of all smart contracts with unique bytecode (i.e., the smart contracts compiled into the same bytecode are considered as one) contain inline assembly and more than 8 million transactions executed the bytecode segments compiled from inline assembly. Moreover, we find that 90% of inline assembly bytecode segments contain no more than 31 instructions and about 88% of inline assembly bytecode segments are executed by no more than 10 transactions. Our answer to **RQ2** reveals that the smart contracts with/without inline assembly differ in five metrics detailed in §3.3. Such a comparison study leads to three insights. First, developers can use some specific opcodes directly through inline assembly to implement functionalities unavailable in Solidity. Second, developers tend to use inline assembly with short length to implement uncomplicated control flows in long smart contracts. Third, developers using inline assembly may pay more attention to the performance of smart contracts than those writing pure Solidity code. We answer **RQ3** by manually examining the inline assembly in all open-source smart contracts and discover three major reasons for using inline assembly, namely realizing functionalities that cannot be implemented by Solidity, producing gas-efficient bytecode, and easing smart contract development. To understand the viewpoints of developers on inline assembly in smart contracts, we also conduct a survey on it and receive valuable feedback from 122 smart contract developers in 22 countries. They are very interested in our study and provide informative feedback and suggestions. For example, they believe that inline assembly can complement Solidity and expect Solidity to provide built-in support to the functionalities that can currently only be implemented through inline assembly. Some practitioners also present concerns for security risks and maintenance difficulties of inline assembly.

In summary, our work has three major contributions:

- We conduct the *first* large-scale empirical study on inline assembly used in smart contracts by analyzing 7.6 million open-source smart contracts from three aspects, namely, source code, bytecode, and transactions.
- We design new approaches to collect open-source smart contracts as many as possible, to extract the genuine bytecode segment compiled from inline assembly, and to determine transactions that invoke inline assembly efficiently. The data set for our empirical study is released at https://github.com/solassem/solassembly.
- We obtain many new observations and insights, functionalities and patterns that are useful to smart contract developers and Solidity developers through quantitative and qualitative analysis on the collected data and a questionnaire survey.

This paper is organized as follows. §2 introduces some background knowledge. §3 presents the methodology of the empirical study. §4 shows the results of the quantitative analysis. §5 presents the results of the qualitative analysis. §6 is about the survey. §7 discusses the threats of our study. §8 briefly retrospects related studies. §9 concludes this paper.

2 Background

Account. Ethereum consists of two kinds of accounts: EOA (External Owned Account) and smart contract account [35]. The latter has bytecode, but the former does not [35].

EVM bytecode. A smart contract should be compiled into bytecode consisting of EVM instructions before being deployed to the blockchain. The Solidity compiler can optimize EVM bytecode if needed. The bytecode consists of two parts: the *constructor* bytecode followed by the *runtime* bytecode [3]. A smart contract can have a constructor, which will be compiled into the constructor bytecode. The constructor bytecode initializes the smart contract during deployment but itself will not be deployed to the blockchain. After deployment, the runtime bytecode is stored on the blockchain and associated with a unique address.

Transaction. A *transaction* is a message that can be used to deploy or invoke smart contracts [35]. To deploy a smart contract, the corresponding transaction carries its constructor bytecode and runtime bytecode [35]. To invoke a smart contract, the transaction should specify the address of the callee, the function to be called, and the parameters [35]. A transaction will fail if an error (e.g., executing an invalid instruction) happens.

Gas. A transaction sender should pay transaction fee because every transaction consumes computing resources of the blockchain. Ethereum uses the *gas* mechanism to calculate the amount of transaction fee. Every EVM instruction has a *gas cost*, the amount of gas that should be paid for executing the instruction, which is roughly proportional to the number of computing resources consumed to execute the instruction [36]. Hence, different EVM instructions can have different gas costs.

Solidity, Yul. Being the most popular high-level language for developing smart contracts, Solidity supports inline assembly [3] through an intermediate language named Yul, which supports several high-level constructs (e.g., switch...case, if, for loops) [34]. Inline assembly can enjoy the optimizations for Solidity, which are applied to EVM bytecode [37], and the optimizations conducted by Yul optimizer, which operates on Yul code [34]. Hence, the bytecode resulted from inline assembly undergoes more optimizations than the bytecode resulted from Solidity code.

Ethereum instruction architecture Ethereum defines more than 130 EVM instructions [36], which differ from traditional instruction sets (e.g., x86/x64) in not only their syntax and semantics but also memory access mode. Specifically, Ethereum allocates a temporary memory space, named *memory* to store temporary values such as local variables, parameters, return values [35]. Ethereum also allocates a permanent memory space, called *storage* to store <key, value> pairs [35]. Ethereum provides special EVM instructions to access memory and storage [36].

3 METHODOLOGY

3.1 Research Questions

To characterize the usage of inline assembly in smart contracts, we investigate the following research questions.

RQ1: How is the prevalence of inline assembly in smart contracts? Besides showing how widespread is inline assembly, the answer inspires the investigation of subsequent research questions. For example, we find that inline assembly is frequently used to implement some functionalities that cannot be realized by Solidity. It motivates us to further examine why inline assembly is used in smart contracts (RQ3).

RQ2: What are the differences between the smart contracts with/without inline assembly in terms of five metrics, including M1 (EVM bytecode length), M2 (frequency of EVM instructions), M3 (is optimized?), M4 (number of transactions invoking a smart contract) and M5 (gas usage)? The answer provides insights for better understanding Ethereum ecosystem and how to help smart contract developers. For example, we find that the smart contracts with inline assembly are more likely to be compiled with optimization, compared to those without inline assembly. It may suggest that the developers using inline assembly pay more attention to the performance of smart contracts than those writing pure Solidity code. With this in mind, we discover how inline assembly can be used to save gas (§5.2). The Solidity compiler can be improved accordingly to help developers reduce gas consumption of their smart contracts.

RQ3: Why do smart contracts use inline assembly? The answer can help developers understand the capability of inline assembly and offer improvement suggestions to Solidity and its compiler. For example, we identify *12* functionalities that can only be implemented by inline assembly. Such findings can guide developers to write powerful smart contracts and are also useful to the designers of Solidity for deciding which new functionalities should be supported in the new version of Solidity. We also observe that inline assembly is used to produce gas-efficient bytecode, which suggests that there is still much room for compilers to optimize the bytecode of smart contracts.

3.2 Data Collection and Preprocessing

We inspect all available open-source smart contracts' source code, bytecode, and transactions to answer the above RQs. Fig. 1 shows the procedure of data collection and preprocessing.

We first collect all open-source smart contracts, whose bytecode has been deployed to Ethereum, with the help of Etherscan [38], which is the most popular Ethereum blockchain explorer, and BigQuery [39], which is a data collection and analysis platform supporting Ethereum. Then, we collect all transactions relevant to these smart contracts by instrumenting EVM. Next, we divide the smart contracts into two groups depending on whether they use inline assembly and compile their source code into bytecode with the same compiler versions and optimization levels as their deployed bytecode.

Given the source code of a smart contract containing inline assembly, after compiling it into EVM bytecode, it is very challenging to recognize the bytecode segments that are compiled from the inline assembly in the whole bytecode. To address the challenge, we design a *novel* approach to extract such bytecode segments by inserting crafted inline statements into the start and the end of each inline assembly fragment in source code. After compiling the source code into bytecode, we can easily find the locations of the instruction sequences compiled from the crafted inline statements. With the help of these locations, we extract the bytecode segments from the original bytecode without insertion. Since it is very difficult, if not impossible, to recognize the bytecode

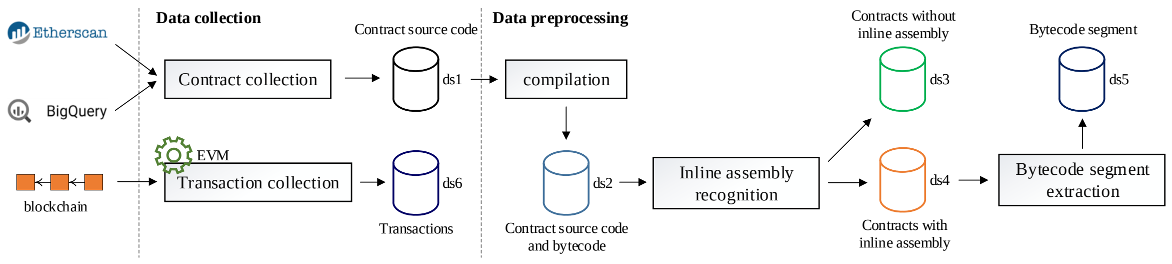
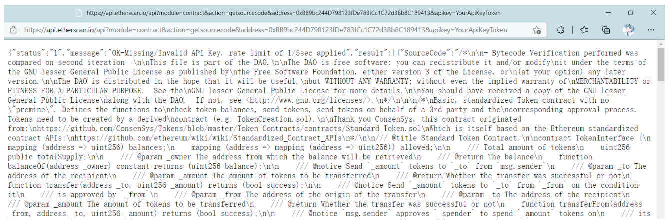


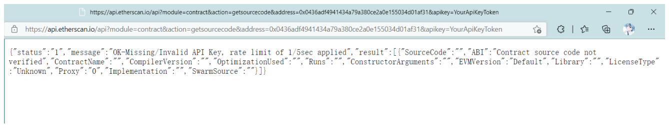
Fig. 1. Data collection and preprocessing


https://api.etherscan.io/api?module=contract&action=getsourcecode&address=0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413&apikey=YourApiKeyToken

https://api.etherscan.io/api?module=contract&action=getsourcecode&address=0xBB9bc244D798123fDe783fCc1C72d3Bb8C189413&apikey=YourApiKeyToken

```
{"status":"1","message":"OK-Missing/Invalid API Key, rate limit of 1/5sec applied","result":[{"SourceCode":"/*\n\n- Bytecode Verification performed was
compared on second iteration -\n\nThis file is part of the DAO.\n\nThe DAO is free software: you can redistribute it and/or modify\nit under the terms of
the GNU lesser General Public License as published by\nthe Free Software Foundation, either version 3 of the License, or\n(at your option) any later
version.\n\nThe DAO is distributed in the hope that it will be useful,\nbut WITHOUT ANY WARRANTY; without even the implied warranty of\nMERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE.  See the\nGNU lesser General Public License for more details.\n\nYou should have received a copy of the GNU lesser
General Public License\nalong with the DAO.  If not, see <http://www.gnu.org/licenses/>.\n*/\n\n\n/*\nBasic, standardized Token contract with no
\"premine\". Defines the functions to\ncheck token balances, send tokens, send tokens on behalf of a 3rd party and the\ncorresponding approval process.
Tokens need to be created by a derived\ncontract (e.g. TokenCreation.sol).\n\nThank you ConsenSys, this contract originated
from:\nhttps://github.com/ConsenSys/Tokens/blob/master/Token_Contracts/contracts/Standard_Token.sol\nWhich is itself based on the Ethereum standardized
contract APIs:\nhttps://github.com/ethereum/wiki/wiki/Standardized_Contract_APIs\n*/\n\n/// @title Standard Token Contract.\n\ncontract TokenInterface {\n
mapping (address => uint256) balances;\n    mapping (address => mapping (address => uint256)) allowed;\n\n    /// Total amount of tokens\n    uint256
public totalSupply;\n\n    /// @param _owner The address from which the balance will be retrieved\n    /// @return The balance\n    function
balanceOf(address _owner) constant returns (uint256 balance);\n\n    /// @notice Send `_amount` tokens to `_to` from `msg.sender`\n    /// @param _to The
address of the recipient\n    /// @param _amount The amount of tokens to be transferred\n    /// @return Whether the transfer was successful or not\n
function transfer(address _to, uint256 _amount) returns (bool success);\n\n    /// @notice Send `_amount` tokens to `_to` from `_from` on the condition
it\n    /// is approved by `_from`\n    /// @param _from The address of the origin of the transfer\n    /// @param _to The address of the recipient\n
/// @param _amount The amount of tokens to be transferred\n    /// @return Whether the transfer was successful or not\n    function transferFrom(address
_from, address _to, uint256 _amount) returns (bool success);\n\n    /// @notice `msg.sender` approves `_spender` to spend `_amount` tokens on\n    /// its
```

Fig. 2. The result of a open-source smart contract returned by Etherscan


https://api.etherscan.io/api?module=contract&action=getsourcecode&address=0x0436adf4941434a79a380ce2a0e155034d01af31&apikey=YourApiKeyToken

https://api.etherscan.io/api?module=contract&action=getsourcecode&address=0x0436adf4941434a79a380ce2a0e155034d01af31&apikey=YourApiKeyToken

```
{"status":"1","message":"OK-Missing/Invalid API Key, rate limit of 1/5sec applied","result":[{"SourceCode":"","ABI":"Contract source code not
verified","ContractName":"","CompilerVersion":"","OptimizationUsed":"","Runs":"","ConstructorArguments":"","EVMVersion":"Default","Library":"","LicenseType"
:"Unknown","Proxy":"0","Implementation":"","SwarmSource":""}]}
```

Fig. 3. The result of a closed-source smart contract returned by Etherscan

segments that are compiled from inline assembly without the help of the aforementioned locations, we did not inspect closed-source smart contracts.

Contract collection. Although Etherscan maintains lists of verified (which are also open-source) smart contracts, the lists just display the latest 10,000 smart contracts that meet the open-source license [38] or the latest 500 open-source smart contracts compiled by the specified compiler. We propose a *new* method to download the source code of all smart contracts whose source code was submitted to Etherscan (include but are not limited to the lists maintained by Etherscan). We first get the addresses of all smart contracts by querying BigQuery through the statement *"SELECT contracts.address FROM 'bigquery-public-data.crypto_ethereum.contracts' as contracts"*. Then, we invoke a web API [40] provided by Etherscan to query the information of each smart contract.

If the smart contract's source code is available, the result contains the source code, the compiler version and the optimization level (shown in Fig. 2); otherwise, these three fields are empty (shown in Fig. 3). By this approach, we obtain the source code of 7,640,317 smart contracts. Note that Etherscan regards a smart contract as open-source when it has the same bytecode as another open-source smart contract. Therefore, although the number of open-source smart contracts published by Etherscan does not exceed 160,000 [41], there are more than 7.6 million open-source contracts, whose addresses are listed in our Github repository. All fetched source code of smart contracts and their addresses are stored in *ds*1 as shown in Fig. 1.

Compilation. We compile each smart contract in *ds*1 into EVM bytecode with the same compiler version and optimization level as its deployed bytecode (the compiler version and optimization level are collected in the previous step, contract collection). The source code and the EVM bytecode of all open-source smart contracts are stored in *ds*2 as shown in Fig. 1.

Inline assembly recognition. A smart contract may have several fragments of inline assembly statements. Each fragment in the Solidity source code must start with the assembly keyword and encloses its inline assembly statements in a pair of curly braces after the keyword [4] (as shown in Fig. 5(a)). Therefore, we recognize all inline assembly fragments in a smart contract by parsing the source code and locating the assembly{...} construct. If a smart contract has at least one inline assembly fragment, we place it in *ds*4, otherwise we put it in *ds*3 as shown in Fig. 1.

Bytecode segment extraction. To avoid confusion, we use two terms, namely inline assembly *fragment* and bytecode *segment*, to refer to the inline assembly statements embedded in Solidity code and the EVM bytecode compiled from the inline assembly fragment, respectively.

Our approach first inserts crafted inline assembly statements into the start and the end of each inline assembly fragment in source code and then uses their bytecode as marks to locate the bytecode segments in the bytecode of the smart contract. The inserted inline assembly statements should meet three requirements. **R1:** they should not affect the data flow and control flow of the original smart contract. Otherwise, our analysis will be biased, because the program logic of the inline assembly fragment may be changed. **R2:** they must make explicit changes to the state of the blockchain. Otherwise, they may be removed by the compiler through optimization. **R3:** their operands should be distinctive so that we can quickly locate the marks among numerous instructions.

After locating the marks in the bytecode, an intuitive way to collect a bytecode segment is to extract the EVM instructions from the bytecode with insertion between the beginning mark and the end mark. However, we find that with the inserted inline assembly statements the bytecode segments may change slightly, because the inserted statements may modify the stack or change the program counters of jump targets although the program logic of the original inline assembly statements is unchanged. To tackle this issue, our approach records the program counters of the two marks in the *bytecode with insertion*, and then extracts the EVM instructions located by these two program counters from the *original bytecode without insertion*. By doing so, we can collect the genuine bytecode segments compiled from the inline assembly fragments, and store them in $ds5$ as shown in Fig. 1.

We use an example shown in Fig. 5 to illustrate the process of bytecode segment extraction. Fig. 5(a) shows the Solidity source code. Lines 4 - 10 are the embedded inline assembly fragment. Please ignore Lines 5 and 9 at this moment. Fig. 5(b) displays the EVM bytecode compiled from Fig. 5(a), which is 150 bytes long. The bytecode segment compiled from the inline assembly is underlined. We can see that there are no marks in the EVM bytecode to recognize the bytecode segment. Fig. 5(c) presents part of the 89 EVM instructions disassembled from the bytecode in Fig. 5(b). The number inside [...] before each instruction is the value of the program counter (PC). The instructions compiled from the inline assembly fragment are also underlined. The Program Counter (PC) refers to the byte offset of a statement in the smart contract bytecode. It is worth noting that in EVM all instructions except `PUSH` instruction do not have immediate data, and the length of each instruction is one byte. Note that the `PUSH`x family of instructions pushes x (1 to 32) bytes elements immediately following the instructions to the top of the stack. For example, `PUSH2` will push two bytes data to the stack. Since the immediate data of the `PUSH2` instruction at 111 is 0x1998, the immediate data takes 2 bytes (i.e., 112 and 113). Therefore, the PC of the next instruction (i.e., `PUSH2` 0x1111) should be 114. Similarly, since the immediate data of the `PUSH2` instruction at 114 is 0x1111, the immediate data takes 2 bytes (i.e., 115 and 116), and thus the PC of the next instruction (i.e., `SSTORE`) is 117.

Line 5 in Fig. 5(a) refers to the assembly statement inserted to the beginning of the inline assembly fragment. This inserted statement satisfies three requirements. **R1:** it writes data to the storage but the original smart contract does not access the storage, and hence the crafted statement does not affect the program logic of the original smart contract. **R2:** it makes explicit changes to the state of the blockchain because the storage is permanent memory space. **R3:** it has two distinctive operands, e.g., 0x1111 and 0x1998. We choose 0x1111 as the storage space randomly. In particular, we collected the address used by the `SSTORE` and `SLOAD` instructions in all smart contracts with inline assembly and found that none of them used the 0x1111 storage space. Moreover, since we only extract the bytecode, we don't need to care about whether the smart contract will occupy the address in future transactions.

Fig. 5(d) and Fig. 5(e) show the bytecode after inserting Line 5 and the corresponding EVM instructions, respectively. The bytecode and the instructions compiled from the inserted statement can be easily located, which are shaded in Fig. 5(d) and Fig. 5(e). In this example, the bytecode and the instructions corresponding to the original inline assembly statements which are modified after insertion are in red. The reason for the modification is that the inserted statement changes the program counters of two jump targets which are pushed onto the stack by Line 125 and Line 135, as shown in Fig. 5(e). We record the starting program counter, i.e., 111, of the inserted instructions. Similarly, we insert the same statement into the end of the inline assembly fragment (Line 9 in Fig. 5(a)). We can see that the starting program counter of the inserted instructions is 140 in Fig. 5(e). Hence, after subtracting the length (i.e., 7) of the bytecode that does not belong to the original smart contract (i.e., the bytecode generated by beginning mark), we obtain the program counter (i.e., 133) of the instruction right after the bytecode segment that should be extracted. Finally, we extract the bytecode segment which starts from the program counter 111 and ends with the program counter 132, as underlined in Fig. 5(c).

```
func opJumpi(pc *uint64, ...) ([]byte, error) {
    pos, cond := stack.pop(), stack.pop()
    if cond.Sign() != 0 { //jump
        if !contract.validJumpdest(pos) {
            return nil, errInvalidJump
        }
        *pc = pos.Uint64()
        //get pc value
        dataCollector.GetTrace("opcode:Jumpi", *pc)
    } else {//fall through
        *pc++
        //get pc value
        dataCollector.GetTrace("opcode:Jumpi", *pc)
    }

    interpreter.intPool.put(pos, cond)
    return nil, nil
}
```

Fig. 4. JUMPI instrucment code

The optimization mechanisms during the compilation may lead to recombination and merging of instruction sequences. To evaluate whether such optimization mechanisms will affect our approach, we first investigate all 30 Yul-based optimization mechanisms from Solidity's official documents [42], including three loop-based optimization mechanisms, ForLoopConditionIntoBody, ForLoopConditionOutOfBody, ForLoopInitRewriter. Since we find that they focus on variables, loops, function calls, etc. (details in the supplementary material [43]), our marks will not be affected by optimization mechanisms. To validate the correctness of our theoretical analysis, we also manually check the impact of each optimization mechanism on inline marks by experiments. According to the patterns of inline assembly fragments that can be optimized by these 30 optimization methods, we design 30 inline assembly fragments and construct 30 test smart contracts containing these inline assembly fragments. Then, we insert marks into the boundaries of these inline assembly fragments in these 30 test smart contracts, and enable the compiler's optimization option. After that, we analyze the bytecode of these smart contracts. Our manual inspection shows that our marks are always located at the beginning and end of the inline assembly fragments without being affected by the optimization.

Oyente [44] and Madmax [45] can locate the location of the bytecode corresponding to the inline assembly by using the source location mapping function provided by the compiler. However, they can not achieve our purpose because of two reasons. First, this feature cannot accurately locate the variables declared in the inline assembly [46]. Second, inline assembly is available after V0.3.1 whereas the source location mappings for bytecode is available after V0.3.6 [47], and thus this feature cannot handle some inline assembly fragments. In contrast, our method can locate all of them correctly.

Transaction collection. Although there are some data collection platforms for Ethereum [38], [39], [48], [49], they cannot provide the information about which instructions are executed by a transaction. Such information is required to determine whether a transaction executes inline assembly. An intuitive approach is used to record all executed instructions by instrumenting all interpretation handlers in the EVM, each of which interprets an EVM instruction. Unfortunately, this approach is inefficient because a number of instructions executed by one transaction.

We propose an *efficient* approach that just records the information of jump instructions by instrumenting the instruction execution part of the go-ethereum client and obtaining relevant information about the instruction, including the program counters of jump instructions and their target instructions, because jump instructions (function calls within a smart contract are compiled into jump instructions [36]) are used for altering the control flow. For example, since `JUMPI` instruction is a conditional jump instruction, and `JUMP` is a direct jump instruction, we modify the `JUMPI` instruction of the EVM of go-ethereum to obtain the corresponding instruction and PC in Fig. 4 (`JUMP` is handled through a similar process). More precisely, we first develop a data record object dataCollector and record the data generated by the Ethereum client when executing the instruction. In Fig. 4, the jump destination is recorded in the handler interpretation function opJumpi() executed by the `JUMPI` instruction (Line 9 and 13). Knowing control flow transfers and the bytecode of a smart contract, we can determine which instructions are executed by a transaction. Ethereum executes all transactions sequentially in each block, and thus we can record the execution result of each transaction by inserting a logging code after it completes execution. Specifically, for each transaction that invokes a smart contract, we record the address of the invoked smart contract, whether the transaction is successful and the gas consumption. Our approach collects 112,326,642 transactions which were sent to smart contracts in *ds*3 and *ds*4 with blocks ranging from 0 to 10,000,000, and stores them in *ds*6 as shown in Fig. 1. Please note that we do not need to record the transactions sent to EOAs, because they do not execute smart contracts.

3.3 Investigation Procedures

RQ1: We analyze the data in *ds*2, *ds*4, *ds*5, and *ds*6 to answer RQ1. First, we count the number of smart contracts with inline assembly by querying *ds*4, and then compute the proportion of such smart contracts to all smart contracts, which are stored in *ds*2. For each smart contract with inline assembly, we count how many inline assembly fragments it has by querying *ds*4. Then we count the number of EVM instructions in each inline assembly segment by querying *ds*5. Moreover, we count the number of transactions executing inline assembly and compute its proportion by querying *ds*5 and *ds*6. For each executed inline assembly segment, we count the number of transactions executing it by querying *ds*5 and *ds*6. Finally, we also conducted the above analysis after removing the toy contracts (i.e., smart contracts that have not been invoked).

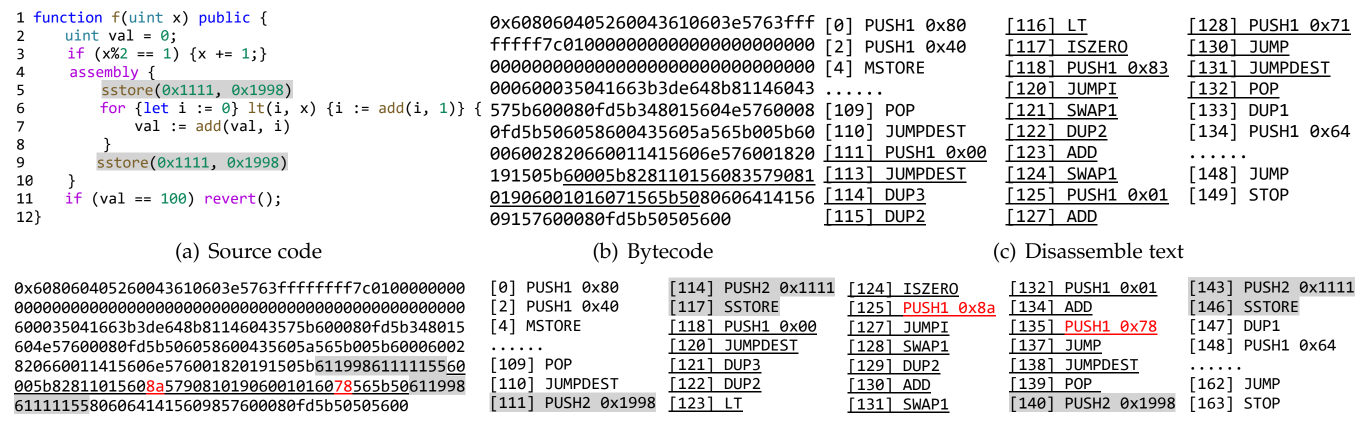
RQ2: We examine the data in *ds*3, *ds*4, *ds*5, and *ds*6 to answer RQ2. Specifically, we define five metrics (**M1**-**M5**) for comparing the smart contracts with and the smart contracts without inline assembly. **M1**-**M3** are static metrics and **M4**-**M5** are dynamic metrics. A *dynamic* metric is only applicable after the execution of smart contracts, e.g., the gas consumption, while a *static* metric has no such restriction, e.g., the length of EVM bytecode. We perform statistical tests to compare two distributions corresponding to two sets of smart contracts ds3 and ds4. More specifically, we apply a two sided Mann-Whitney U test (which is equivalent to the Wilcoxon rank-sum test [50]) to investigate whether two populations have the same distribution [51] and consider two distributions are different at $\alpha = 0.05$.

We regard smart contracts that have not been invoked as toy contracts, and smart contracts that have been invoked at least once as non-toy contracts. For the static metrics **M1**-**M3**, we study both non-toy and toy contracts.

M1: EVM bytecode length, which is an indicator of the complexity of smart contracts. Since there are no standard criteria for measuring the complexity of different programs, we use the length of bytecode to measure the complexity of smart contracts for the following reason. Nystedt et al. [52] suggested that the line of code could be used as a measure. Since the source code of smart contracts may not be available, we use the length of bytecode, which will be executed by EVM, as an approximation to the line of code to measure the complexity of smart contracts. We also used SolMet [53] to compute the complexity of the smart contracts with/without inline assembly and selected commonly used complexity metrics SLOC (number of source code lines) and WMC (weighted sum of McCabe's style complexity over the functions of a contract) for analysis (§4.2).

M2: frequency of EVM instructions. It serves as an indicator of the functionalities of smart contracts because we can infer the functionalities of smart contracts according to the semantics of the instructions used by the smart contracts. These functionalities are all low-level logic, but multiple such low-level functions can be combined into high-level business logic. For example, if a smart contract does not contain the EVM instruction `NUMBER`, it cannot read the block number because `NUMBER` is the only instruction to read the block number [36]. Therefore, **M2 (frequency of EVM instructions)** can reflect the difference of functionality between inline assembly and Solidity. To compare the functionalities of inline assembly fragments with those implemented in Solidity, we count the number of each EVM instruction appearing in the bytecode stored in *ds*5 and *ds*3, individually. We ignore the following instructions, `PUSH`x $1 \leq x \leq 32$ for pushing a number on the stack, `POP` for popping the top stack item, `DUP`x $1 \leq x \leq 16$ for duplicating one stack item, `SWAP`x $1 \leq x \leq 16$ for exchanging two stack items [36], because these stack operations are frequently used in all smart contracts and they do not provide hints on the functionalities of smart contracts.

M3: is optimized? It reflects whether the developers intend to optimize the performance of smart contracts. Since the optimization level of each open-source smart contract is available (§3.2), we count the number of smart contracts with and smart contracts without optimization in *ds*3 and *ds*4, individually.



```
function f(uint x) public {
    uint val = 0;
    if (x%2 == 1) {x += 1;}
     assembly {
        sstore(0x1111, 0x1998)
        for {let i := 0} lt(i, x) {i := add(i, 1)} {
            val := add(val, i)
        }
       sstore(0x1111, 0x1998)
    }
    if (val == 100) revert();
}
```

(a) Source code

```
0x6080604052600043610603e5763fff
fffff7c01000000000000000000000000
00000000000000000000000000000000
00060003504l663b3de648b81146043
575b600080fd5b348015604e5760008
0fd5b5060586004356 05a565b005b60
00600282066001141560 6e576001820
191505b60005b82811015608357908l
01906001016071565b5080606414156
09157600080fd5b50505600
```

(b) Bytecode

```
[0] PUSH1 0x80        [116] LT             [128] PUSH1 0x71
[2] PUSH1 0x40        [117] ISZERO         [130] JUMP
[4] MSTORE            [118] PUSH1 0x83     [131] JUMPDEST
......                [120] JUMPI          [132] POP
[109] POP             [121] SWAP1          [133] DUP1
[110] JUMPDEST        [122] DUP2           [134] PUSH1 0x64
[111] PUSH1 0x00      [123] ADD            ......
[113] JUMPDEST        [124] SWAP1          [148] JUMP
[114] DUP3            [125] PUSH1 0x01     [149] STOP
[115] DUP2            [127] ADD
```

(c) Disassemble text

```
0x6080604052600043610603e5763ffffffff7c0100000000
0000000000000000000000000000000000000000000000000
600035041663b3de648b81146043575b600080fd5b348015
604e57600080fd5b5060586004356 05a565b005b60006002
82066001141560 6e57600182019150 5b611998611111 5560
005b8281101560 8a57908101906001016078565b50611998
61111155806064141560985 7600080fd5b50505600
```

(d) Bytecode after inline assembly insertion

```
[0] PUSH1 0x80        [114] PUSH2 0x1111   [124] ISZERO         [132] PUSH1 0x01     [143] PUSH2 0x1111
[2] PUSH1 0x40        [117] SSTORE         [125] PUSH1 0x8a     [134] ADD            [146] SSTORE
[4] MSTORE            [118] PUSH1 0x00     [127] JUMPI          [135] PUSH1 0x78     [147] DUP1
......                [120] JUMPDEST       [128] SWAP1          [137] JUMP           [148] PUSH1 0x64
[109] POP             [121] DUP3           [129] DUP2           [138] JUMPDEST       ......
[110] JUMPDEST        [122] DUP2           [130] ADD            [139] POP            [162] JUMP
[111] PUSH2 0x1998    [123] LT             [131] SWAP1          [140] PUSH2 0x1998   [163] STOP
```

(e) Disassemble text after inline assembly insertion

Fig. 5. Example of bytecode segment extraction

M4: number of transactions invoking a smart contract. It may reflect the popularity of a smart contract because popular smart contracts will attract more users. We compute the number of transactions invoking each smart contract in *ds*4 and *ds*3, respectively, with the help of *ds*6, which stores the destination addresses of all transactions.

M5: gas usage, which may reflect the popularity and complexity of a smart contract, because executing a complex smart contract will consume more gas and a popular smart contract may attract more transactions and thus lead to more total gas usage. For example, we found two real smart contracts with differences in popularity and complexity. A smart contract named SafeConditionalHFTransfer [54] contains only 31 lines of Solidity code, and thus it is a fairly simple smart contract. However, due to its popularity (called by 20,772 transactions), it consumes a total of 622,091,325 gas. Another smart contract named SGTExchanger [55] is unpopular and involves only 961 transactions, but it is relatively complex (1,847 lines of source code). It consumes a total of 149,890,879 gas. Both smart contracts consume a lot of gas.

RQ3: To discover the reasons why inline assembly is used, four authors of this paper manually investigate *all* inline assembly fragments in *ds*4. Fig. 6 shows the process of our manual analysis. For each inline assembly fragment we obtained by matching the assembly keyword from the source code, in order to try to use Solidity to achieve the same functionality, we first learn its functionality by reading the inline assembly statements and the corresponding comments (if existed). Then, to better understand its execution process, we generate the corresponding mock data in advance. That is, the parameters of the corresponding function are generated through the list of function parameter types provided by its ABI. After that, we deploy the contract with the inline assembly fragment to Remix [56] for manual simulation execution, that is, use mock data to initiate a transaction to the contract and then debug the transaction that executes the function to view the stack change during the instruction execution, and then compare it with the functionality of the corresponding instruction [36].

To study whether this functionality can only be implemented using inline assembly and whether using inline assembly can save gas, we try to implement the same functionality using Solidity. The authors of this paper have 1-4 years of experience in developing Ethereum smart contracts. If there are different implementations to achieve the same functionality in Solidity, we choose the implementation consuming the least gas. For a fair comparison, we compile each implementation in Solidity using the same compiler version and optimization level as those used to produce the corresponding inline assembly fragment. If we can't use Solidity to achieve it (§5.1), we consider the functionality as the one that cannot be implemented by Solidity code.

Moreover, we use the debugging function on the Remix to analyze instruction execution step by step and investigate why the inline assembly fragment can save gas. To compare the gas consumption between an inline assembly fragment and the bytecode compiled from its Solidity implementation, we deploy the two smart contracts (i.e., smart contract with inline assembly and its Solidity implementation) on Remix, and then call them respectively to obtain the real gas consumption. The limitations and threats of Remix are shown in detail in §7. If it cannot save gas with the inline assembly fragment, we analyze whether the use of inline assembly can ease development. That is, by consulting programming books to determine whether the use of inline assembly can make the programming process more readable and easy.

4 Quantitative Results

4.1 Results for RQ1

Results are shown in Fig. 7. We collect 7,640,317 open-source smart contracts with 78,642 unique EVM bytecode. Among all open-source smart contracts, we find 1,480,947 containing inline assembly fragments. These 1,480,947 smart contracts correspond to 11,198 unique bytecode, and we find 9,355 out of them contain the inline assembly bytecode segments. That is, the inline assembly fragments in 1,843 ($16.5\% = 1,843/11,198$) unique smart contracts are not compiled into bytecode. After reading the source code, we find that these inline assembly fragments are unreachable (e.g., in never-called functions, on infeasible paths). Among all 78,642 unique EVM bytecode, we find 35,178 of them have not been invoked and 4,986 out of them contain inline assembly bytecode segments. Hence, the EVM bytecode

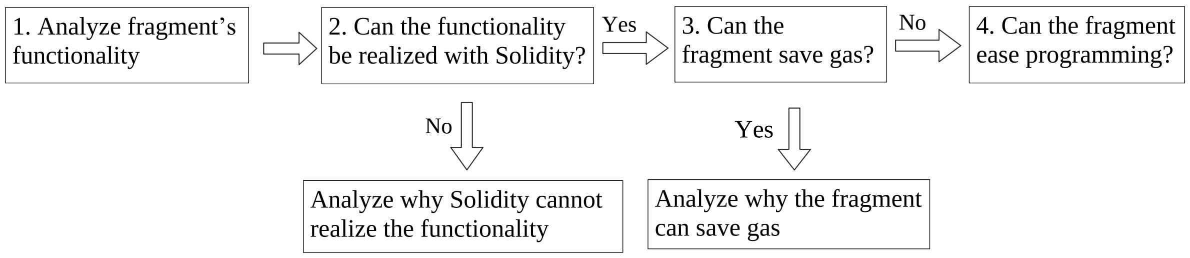

1. Analyze fragment's functionality
2. Can the functionality be realized with Solidity?
Yes
3. Can the fragment save gas?
No
4. Can the fragment ease programming?
No
Analyze why Solidity cannot realize the functionality
Yes
Analyze why the fragment can save gas

Fig. 6. The process of inline assembly fragment analysis.

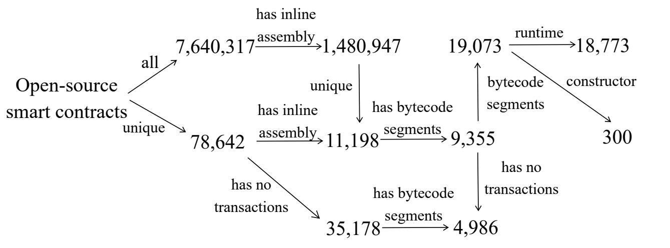


Fig. 7. Results for the prevalence of inline assembly in open-source smart contracts

Year	2016	2017	2018	2019	2020(until March)
# of SCs w/ inline	103	1392	21880	226707	49853
# of all SCs	75490	1304084	3513930	2638212	108130
% of SCs w/ inline	0.14%	0.11%	0.62%	8.6%	46.1%

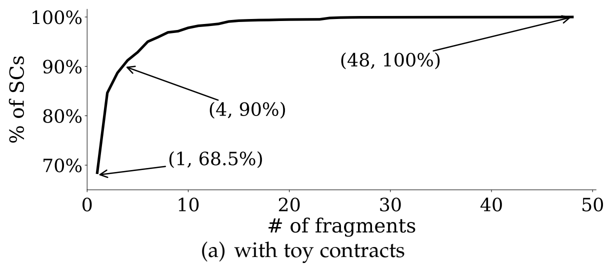
TABLE 1
Number of smart contracts with inline assembly created by year

containing inline assembly bytecode segments accounts for 11.9% $(9,355/78,642)$ of all unique EVM bytecode. And the EVM bytecode of smart contracts that have not been called accounts for 44.7% $(35,178/78,642)$ of all unique EVM bytecode. The EVM bytecode cotaining inline assembly bytecode segments that have not been invoked accounts for 53.3% $(4,986/9,355)$ of all unique EVM bytecode containing inline assembly bytecode segments. From all 9,355 EVM bytecode, we discover 19,073 bytecode segments compiled from inline assembly. Among these segments, 300 (1.6%) exist in the constructor bytecode, and 18,773 (98.4%) exist in the runtime bytecode. After removing toy contracts, the number of open-source contracts with the unique EVM bytecode dropped to 43,464, of which 4,270 contained inline assembly bytecode segments, accounting for 9.8% (4,270/43,464), and this ratio is 14.5% (5,085/35,178) in toy smart contracts. Among the 4,270 EVM bytecodes, there are 8,192 bytecode segments compiled from inline assembly. Of these segments, 115 (1.4%) are in the constructor bytecode, and 8,077 (98.6%) are in the runtime bytecode.

To better understand the use of smart contracts with inline assembly, we investigated the creation time of all open-source smart contracts. As shown in Table 1, the amount of newly created smart contracts with inline assembly is increasing year by year. Specifically, in 2016 and 2017, only 103 and 1,392 smart contracts with inline assembly were created, which increased to 21,980 in 2018 and 226,707 in 2019. As of March 20, 2020, 49,853 smart contracts with inline assembly have been created in 2020, which exceeds the number in the same period in 2019. Then we calculated the proportion of smart contracts with inline assembly in all open-source smart contracts. 0.14% of all open-source smart contracts created in 2016 contain inline assembly, which increased to 0.11%, 0.62% and 8.6% from 2017 to 2019, respectively. In 2020, the proportion reached 46.1%, which is much higher than that in previous years. The reason for such growth is that a smart contract named AccountCreator [57] frequently created smart contracts with inline assembly in 2020. In more detail, AccountCreator was created on December 30, 2019, and then it created a large number of (i.e., 11,476) smart contracts with inline assembly called AccountProxy [58] before March 20, 2020. Similarly, another smart contract with inline assembly called DSProxy [59] had been created 10,160 times. These creations make the newly created smart contracts with inline assembly account for a high proportion of all newly created open-source contracts in 2020. In general, the creation frequency of smart contracts with inline assembly is increasing year by year, demonstrating that the functionalities provided by inline assembly are very useful for smart contract developers.

Insight 1: The tools for analyzing the source code of smart contracts should also handle inline assembly, because 11.9% of smart contracts use it. However, to the best of our knowledge, no source code analysis tool (e.g., VerSol [60], SmartCheck [61], SolidityCheck [62]) can handle inline assembly.

Fig. 8(a) and Fig. 8(b) depict the cumulative distribution function (CDF) of the number of unique inline assembly fragments of the smart contracts with and the smart contracts without toy contracts, respectively from 1 to 48 and from 1 to 33. 68.5% of smart contracts with toy contracts just contain 1 fragment, while those without toy contracts are 71.2%, and 90% of smart contracts with toy contracts contain no more than 4 fragments, while those without toy contracts contain no more than 3 fragments. By inspecting its source code, we find that the smart contract [63] contains 45 functions, and each contains at least one fragment. We also find that the smart contract [64] contains many libraries, and each contains several fragments. These smart contract leverages inline assembly to access arbitrary memory locations and arbitrary storage locations, which cannot be



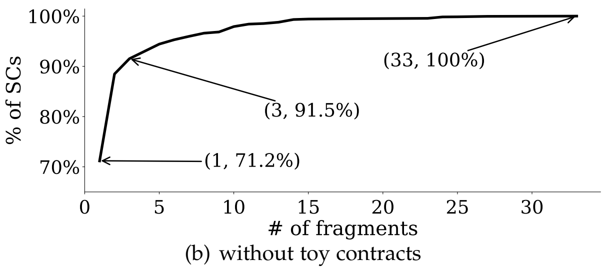
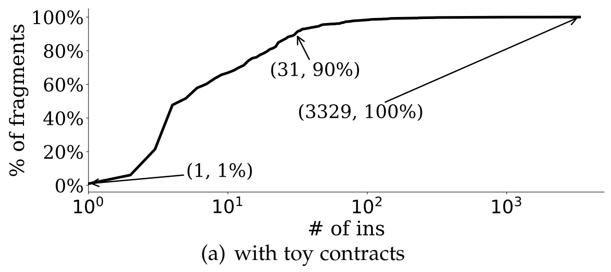


Fig. 8. The number of inline assembly fragments

realized by Solidity (§5.1).



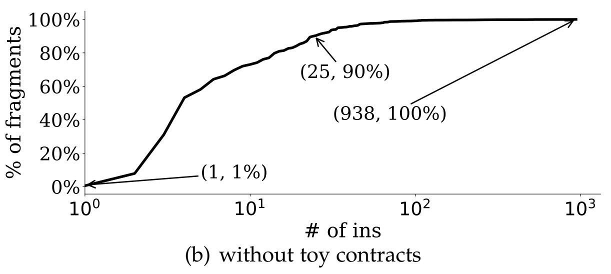
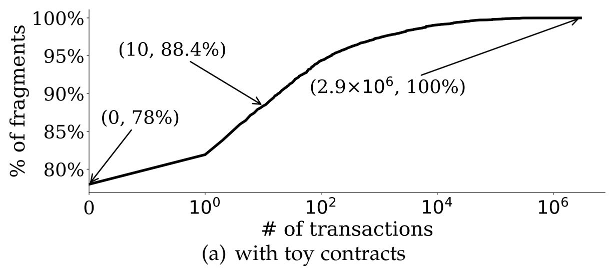


Fig. 9. The number of instructions in inline assembly fragments

Fig. 9(a) and Fig. 9(b) shows the CDF of the number of EVM instructions in each inline assembly bytecode segment of the smart contracts with and the smart contracts without toy contracts. The number ranges from 1 to 3,329 and from 1 to 938, and most segments are short. Fig. 9(a) shows that 90% of the segments in these contracts with toy contracts contain no more than 31 instructions, while Fig. 9(b) shows that 90% of the segments in these contracts without toy contracts contain no more than 25. That is, the very complex segments are not actually invoked and generally, the segments used frequently are relatively short. However, we observe that a segment contains 3,329 instructions. By inspecting the source code, we find that the corresponding smart contract [65] uses inline assembly to implement Poseidon hash function. Poseidon hash function is used for the implementation of Zero-knowledge proof [66] in the Loopring protocol [67], which is a decentralized token exchange protocol.



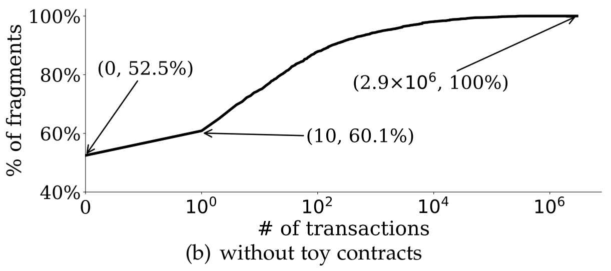


Fig. 10. The number of transactions for each inline assembly fragment

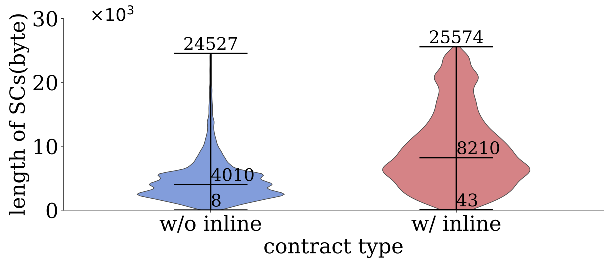
Among all 112,326,642 transactions, 13% (14,583,890) of them invoke the smart contracts containing inline assembly bytecode segments. 58.3% (8,496,835) out of them execute inline assembly code, but 6,785,879 transactions (80%) execute just 123 ($0.7\% = 123/18,773$) segments. Fig. 10(a) and Fig. 10(b) shows the CDF of the number of transactions that execute an inline assembly bytecode segment in the smart contracts with and the smart contracts without toy contracts. Fig. 10(a) shows that about 78% of segments have not been executed yet, and nearly 88.4% of segments have been executed by no more than 10 transactions. The reason for the high proportions is that some popular smart contracts attract most transactions. For instance, one segment has been executed about 2.9 million times. We find that this segment reads parameters by an inline assembly operation `CALLDATALOAD` [4]. It belongs to a function modifier that is used in 11 functions of the EOS token contract. Hence, the segment will be executed whenever one of these 11 functions is invoked. Note that EOS, a popular blockchain, uses Ethereum to host its tokens before launching its mainnet. And Fig. 10(b) shows that after removing the toy contract, 52.5% of the segments have not been executed, and 60.1% of the segments have been executed by no more than 10 transactions. It can be observed that these two proportions decrease after removing the toy contract because the toy contract has not been called by any transaction.

We observe that millions of transactions execute inline assembly fragments, while a few fragments attract most transactions. After investigating the functions of these inline assembly fragments that are not executed or rarely executed, we find that they are both in one of the following two cases: (1) they are in unpopular smart contracts, which have a small number of transactions; (2) they are in the functions of smart contracts which are called infrequently, such as the initialization function of a smart contract, which is called only once during the deployment of the smart contract. Therefore, the execution of the inline assembly fragments is related to the popularity of smart contracts where the fragments are located. Similar to the fact that most transactions on Ethereum are relevant to a small number of smart contracts [68], the execution of inline assembly fragments also shows the long tail phenomenon. That is, most transactions execute a small part of inline assembly fragments.

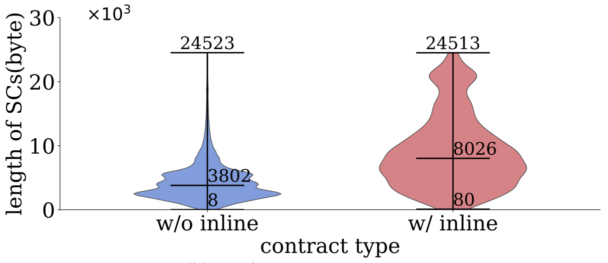
Answer to RQ1: 11.9% of smart contracts use inline assembly and 8.5 million transactions executed inline assembly.

4.2 Results for RQ2

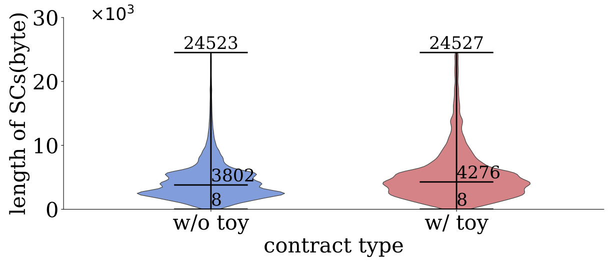
By comparing the smart contracts with and the smart contracts without inline assembly, we find that these two kinds of smart contracts have differences in terms of *all* five metrics.



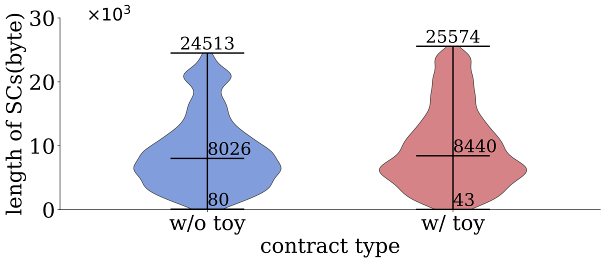
(a) with toy contracts



(b) without toy contracts



(c) without inline assembly



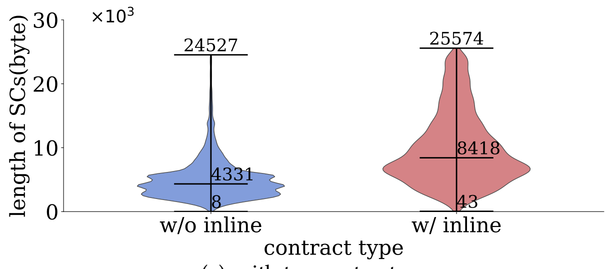
(d) with inline assembly

Fig. 11. The length of bytecode

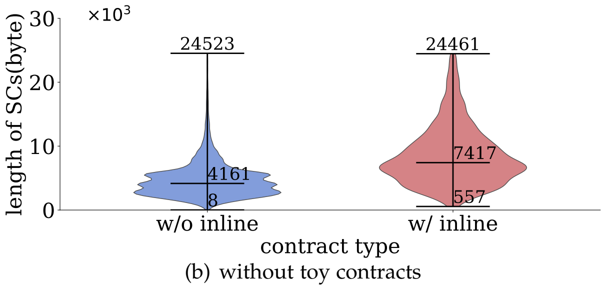
M1 (EVM bytecode length): The violin plot in Fig. 11(a) and Fig. 11(b) show the bytecode length of the *ds*3 and *ds*4 with and without toy contracts, including toy contracts ranging from 8 bytes to 24,527 bytes and 43 bytes to 25,574 bytes, excluding toy contracts, from 8 bytes to 24,523 bytes and 80 bytes to 24,513 bytes, respectively. We also compared the toy smart contracts and the non-toy smart contracts in *ds*3 and *ds*4. Fig. 11(c) and Fig. 11(d) show whether it is *ds*3 or *ds*4, the bytecode length of toy smart contracts is longer than that of non-toy smart contracts, which also shows that toy smart contracts are not simpler, and it is consistent with the difference after removing toy smart contracts in §4.1. We analyzed the source code and its transactions, and found smart contracts that use the factory design pattern, such as the smart contract MultiSigFactory [69], which creates many MultiSigStub contracts through transactions, and their codes are almost the same. And some of them have no transactions like the smart contract [70], while others have a small number of transactions like the smart contract [71]. This is also a possible reason why toy smart contracts are not simple. We performed a Mann-Whitney U test on the EVM bytecode length of the contract in *ds*3 and *ds*4. The input is two arrays. Array A is the EVM bytecode length of the smart contracts with inline assembly, and array B is the EVM bytecode length of the smart contract without inline assembly. The output shows that these two distributions are different. On average, the smart contracts with inline assembly are larger than those without inline assembly. The probability distribution of the right violin is more smooth than the probability distribution of the left violin. Because in *ds*3, most values are close to the median while *ds*4 is the opposite. It means that the average and median length of the bytecode of smart contracts with inline assembly is greater than that of smart contracts without inline assembly. Therefore, the smart contracts with inline assembly are in general more complicated than those without inline assembly. Here are the explanations. A recent study discovers that many toy smart contracts have been deployed on the blockchain [72], which may be adapted from the examples of Ethereum documents and tutorials. These toy smart contracts do not use inline assembly. Contrarily, when developers write inline assembly, which has a higher learning curve compared to Solidity, they may intend to implement special functionalities (c.f., §5.1) in smart contracts for practical applications.

In order to more comprehensively show the difference between *ds*3 and *ds*4 under a certain category, we further conducted experiments for a specific category. Since there is no recognized classification of contracts yet, in order to explore the difference in a certain category, we analyzed all Token contracts and obtained the same observation as the previous one. That is, the bytecode of the Token smart contract with inline assembly will be longer. As shown in Fig. 12(a) and Fig. 12(b), after removing the toy smart contract, compared the Token bytecode length of *ds*3 and *ds*4, it can be seen that the conclusion is consistent with the above, and the length of the token contract with inline assembly is longer. Similarly, Fig. 12(c) and Fig. 12(d) also show that in the category of Token, the bytecode length of toy smart contracts is still longer than that of non-toy smart contracts.

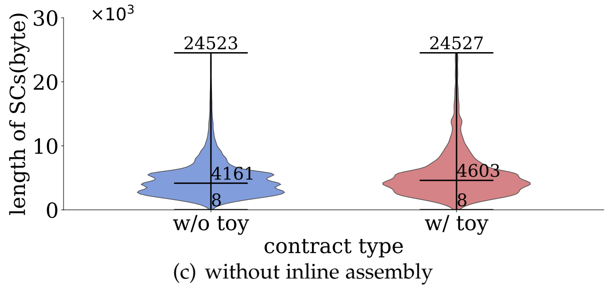
We also used SolMet [53] to compute the complexity of the smart contract with inline assembly and the smart contracts without inline assembly, and selected two com-



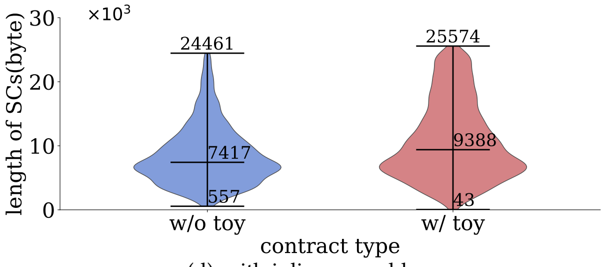
(a) with toy contracts



(b) without toy contracts

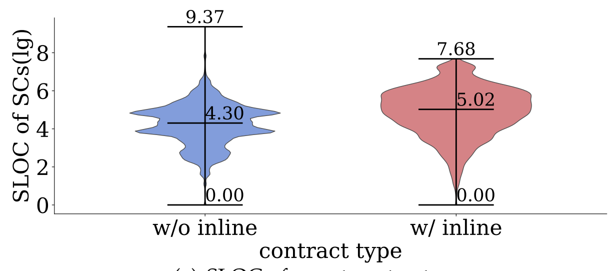


(c) without inline assembly

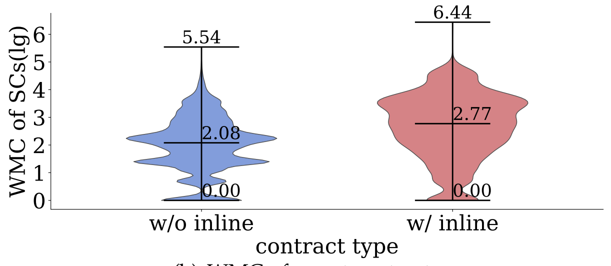


(d) with inline assembly

Fig. 12. The length of token bytecode



(a) SLOC of smart contracts

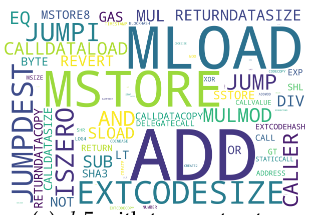


(b) WMC of smart contracts

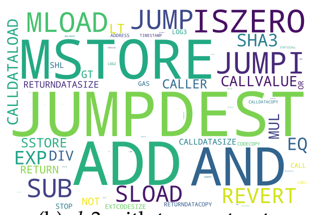
Fig. 13. The complexity of smart contracts

monly used complexity metrics SLOC (number of source code lines) and WMC (weighted sum of McCabe's style complexity over the functions of a contract) for analysis. The former is often used to measure the volume of a program, and it can also measure the complexity of a program, while the latter is a quantitative measure of the number of linearly independent paths through a program's source code [73]. Fig. 13(a) and Fig. 13(b) shows the difference between *ds*3 and *ds*4 in the metrics SLOC and WMC. From the median point of view, the smart contracts with inline assembly are more complex in terms of SLOC and WMC, and their graphics are smoother, which is also consistent with the results obtained by above bytecode length.

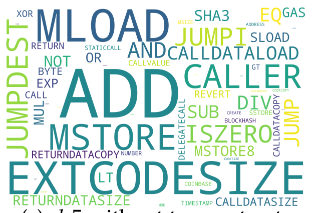
M2 (frequency of EVM instructions)): Fig. 14(a) and Fig. 14(b) present the opcode clouds of the EVM instructions in inline assembly bytecode segments and of those compiled from Solidity code, respectively. Fig. 14(c) and Fig. 14(d) present the opcode clouds of the EVM instructions in inline assembly bytecode segments and of those compiled from Solidity code after removing toy contracts, respectively. We performed a Mann-Whitney U test on the frequency of instruction of the contract in *ds*3 and *ds*5. The input is two arrays. Array A is the frequency of each instruction of the inline assembly segments, and array B is the frequency of each instruction of the smart contract without inline assembly. The output shows that these two distributions are different. We can find many differences between the two opcode clouds with and the two opcode clouds without toy contracts, indicating that inline assembly has special capabilities that cannot be substituted by Solidity.



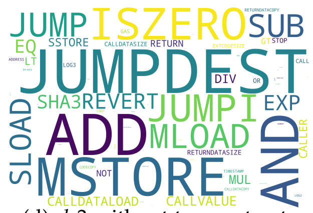
(a) *ds*5 with toy contracts



(b) *ds*3 with toy contracts



(c) *ds*5 without toy contracts



(d) *ds*3 without toy contracts

Fig. 14. Opcode clouds

(1) `EXTCODESIZE` frequently appears in inline assembly segments and rarely in the bytecode compiled from Solidity. The reason is that `EXTCODESIZE` is used for obtaining the runtime bytecode size of a given account, and such functionality can only be realized by inline assembly (§5.1). We find that this instruction only appears in the bytecode compiled from Solidity in a rare circumstance, that is, instantiating the

```
1  contract B {
2     function f() public returns(uint) {
3        return 1024;
4     }
5  }
6
7  contract C {
8     function f(address _addr) public returns(uint) {
9        B b = B(_addr);
10       return b.f();
11    }
12 }
```

Fig. 15. Generate EXTCODESIZE using Solidity

contract object. Fig. 15 shows how to generate the instruction through Solidity. We instantiated a *B* contract in contract *C* through the passed-in address (Line 9), and the compiler used the `EXTCODESIZE` instruction in this process. Since this instruction cannot be used explicitly, `EXTCODESIZE` rarely appears in bytecode compiled from Solidity.

Insight 2: Developers can use some specific opcodes directly through inline assembly to implement functionalities unavailable in Solidity.

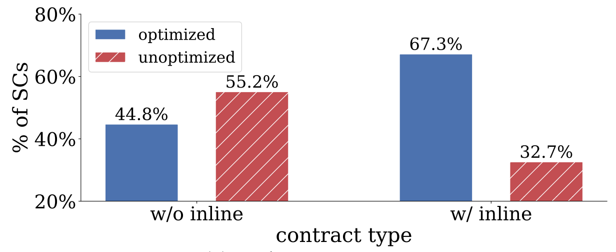
(2) `MLOAD` appears more frequently in inline assembly segments than in bytecode compiled from Solidity. The possible reason is that using `MLOAD` in inline assembly can read arbitrary memory locations, which can not be implemented through Solidity. For example, we find that three functionalities (i.e., F1, F8, F10 in §5.1) not supported by Solidity use the `MLOAD` instruction to read memory at a specific location. 45.2% of inline assembly segments contain `MLOAD` and the inline assembly in smart contracts generates fewer instructions than the Solidity code, which may also be the reason for the higher frequency of `MLOAD`.

(3) `JUMPDEST`, `JUMPI` and `ISZERO` appear more frequently in the bytecode compiled from Solidity than in inline assembly segments. Note that these three instructions are the results compiled from control flow statements (e.g., if...else, for{...} loop). Therefore, it suggests that developers do not tend to write complex control flow transfers in inline assembly. Such finding is accordant to Fig. 9 that most inline assembly byte segments are short.

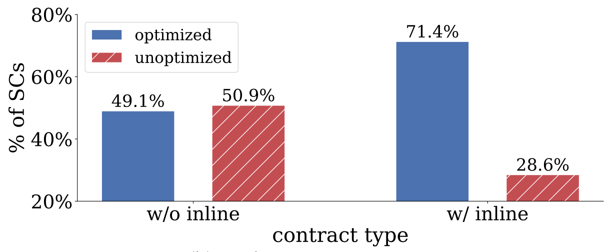
Insight 3: Developers tend to use inline assembly with short length to implement uncomplicated control flows in long smart contracts.

(4) `AND` appears more frequently in the bytecode compiled from Solidity than in inline assembly segments. The frequency of `AND` appearing in the bytecode generated by Solidity is 8.54%, which is a high frequency because there are 142 opcodes in total [36]. The reason is that the compiler automatically adds `AND` to ensure the validity of parameters, and in some arithmetic operations, the compiler also generates `AND` instructions. But the compiler will not automatically generate `AND` in the inline assembly. For example, when a function takes in an 8-bit unsigned integer, the 32-byte data read from a transaction will be masked with 0xff using `AND` to retain the lowest 8 bits. By contrast, we find that inline assembly tends to use arguments directly without masking.

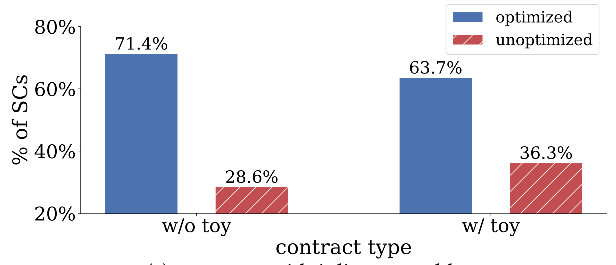
As shown in Fig. 14(c) and Fig. 14(d), the instruction frequency distribution has no change after removing the toy smart contracts because the instruction frequency distribution of toy smart contracts is very similar to that of non-toy smart contracts.



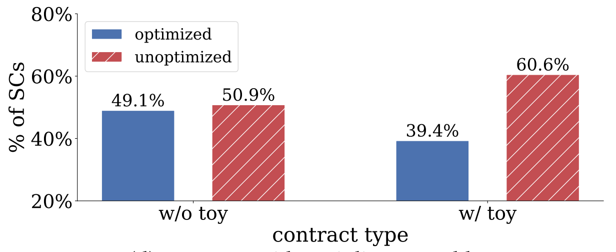
(a) with toy contracts



(b) without toy contracts



(c) contracts with inline assembly



(d) contracts without inline assembly

Fig. 16. The proportion of optimized smart contracts

M3 (is optimized?): Fig. 16(a) and Fig. 16(b) show the proportion of smart contracts with and the proportion of smart contracts without toy contracts which are compiled with optimization. We performed a Phi-coefficient [74] correlation analysis between the contracts in *ds*3 and *ds*4 and the optimization. We build a 2 x 2 table (optimized/non-optimized, contract type) and calculated its Phi coefficient. The results are that the contract in *ds*3 with toy contracts and the optimization have Phi-coefficient $|\phi| = 0.14$, and the contract in *ds*3 without toy contracts and the optimization have Phi-coefficient $|\phi| = 0.13$, both have a weak positive correlation. We can see that about 45% of the smart contracts without inline assembly are optimized, while the proportion increases to about 67% for the smart contracts

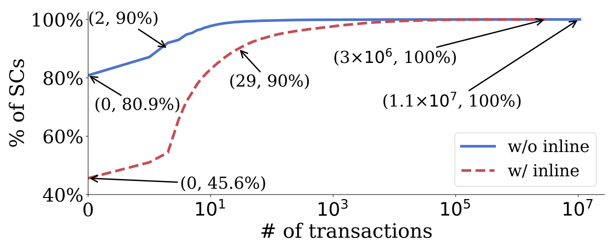


Fig. 17. The number of transactions invoking smart contracts

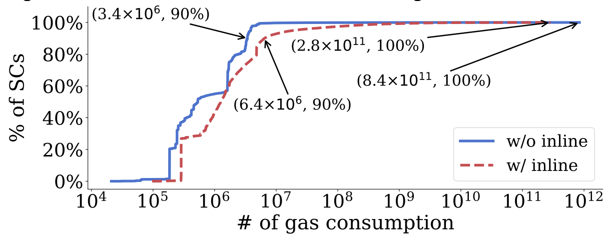


Fig. 18. Gas usage of smart contracts

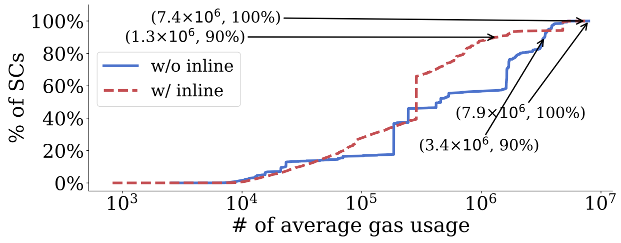


Fig. 19. Average gas usage of smart contracts

with inline assembly. After removing the toy contracts, the optimization proportion increased to 49.1% and 71.4%, respectively. This result suggests that the smart contracts with and without inline assembly differ in optimization and the developers using inline assembly pay more attention to the performance of smart contracts than those writing pure Solidity code. Such finding is accordant with §5.2 that inline assembly can be used to save gas. We also compared the toy smart contracts and the non-toy smart contracts in *ds*3 and *ds*4. Fig. 16(c) and Fig. 16(d) show no matter in *ds*3 or *ds*4, the optimization ratio of toy smart contracts is lower than that of non-toy smart contracts. This also may be a reason that why the bytecode of toy smart contracts is longer.

Insight 4: Developers using inline assembly may pay more attention to the performance of smart contracts than those writing pure Solidity code.

M4 (number of transactions invoking a smart contract): Fig. 17 displays the CDF of the number of transactions calling smart contracts. We performed a Mann-Whitney U test on the number of transactions of the contract in *ds*3 and *ds*4. The input is two arrays. Array A is the number of transactions of the smart contracts with inline assembly, and array B is the number of transactions of the smart contracts without inline assembly, the output shows that these two distributions are different. The curve of the smart contracts with inline assembly is below that of the smart contracts without inline assembly, indicating that the former attracts more transactions. Particularly, about 45.6% of the smart contracts using inline assembly have not been invoked, and this proportion soars to 80.9% for the counterpart. 90% of smart contracts without inline assembly are invoked by no more than 2 transactions. By contrast, 90% of smart contracts with inline assembly are invoked by no more than 29 transactions. We observe that the smart contracts with inline assembly have more transactions on average than the smart contracts without inline assembly. The reason may be that the smart contracts with inline assembly are usually used for practical applications, which are more likely to be frequently invoked. However, among the smart contracts without inline assembly, there are many toy smart contracts that are never invoked.

In addition, we studied whether the smart contracts with or without inline assembly would lead to more or less failed transactions. Please see the supplementary material [43] for details.

M5 (gas usage): Fig. 18 presents the CDF of the gas consumption of each smart contract, which is the summation of the gas consumption of all transactions invoking that smart contract. We performed a Mann-Whitney U test on the gas consumption of the contract in *ds*3 and *ds*4. The input is two arrays. Array A is the gas consumption of the smart contracts with inline assembly, and array B is the gas consumption of the smart contracts without inline assembly, the output shows that these two distributions are different. We observe that the smart contracts with inline assembly consume more gas than the counterpart, which is consistent with our previous observation that the smart contracts with inline assembly are in general more complicated and have more transactions invoked them.

We have also made statistics on the average gas consumption of each smart contract. Fig. 19 presents the CDF of the average gas consumption of each smart contract. The average gas consumption of 90% of smart contracts with inline assembly is lower than 1.3x10^6, while the average gas consumption of smart contracts without inline assembly is lower than 3.4x10^6, which means that smart contracts without inline assembly consume more gas than smart contracts with inline assembly. And the average maximum gas consumed by smart contracts with inline assembly is 7.4x10^6, and this number is 7.9x10^6 for the counterpart.

Answer to RQ2: The smart contracts with and without inline assembly differ in all five metrics.

In addition to the above comparison of the smart contracts with and without inline assembly, we also derive three observations from the analysis of toy smart contracts. First, toy smart contracts are very complex, and their bytecode is longer than non-toy smart contracts, which may be caused by a lower optimization ratio. Second, the proportion of toy smart contracts containing inline is even higher (14.5%), and this proportion is 9.8% in non-toy smart contracts. Third, toy smart contracts contain more inline assembly fragments and inline assembly fragment instructions than non-toy smart contracts.

5 Qualitative Results

We find three major reasons for using inline assembly in smart contracts. First, inline assembly is used for implementing 12 unavailable functionalities in Solidity. More complex functionalities can be realized based on these 12 functionalities. For example, to obtain the bytecode hash of a given account, we can first copy the bytecode of the account (**F8**, §5.1) and then compute its hash by the function keccak256() supported in Solidity. Note that we only enumerate the 12 basic functionalities in §5.1 instead of such composite functionality. The fact that Solidity did not implement these functionalities may be due to security considerations, but the security and functionalities are tradeoffs, and our survey results in §6 show that a few developers have security concerns, but most of them still hope that Solidity has these functionalities. Second, inline assembly can be used for producing gas-efficient bytecode by any of 6 programming patterns (§5.2), which costs less transaction fee than the counterpart in Solidity. Third, inline assembly has special programming constructs, which can ease development (§5.3), even if Solidity source code can be compiled into the same bytecode.

5.1 Supporting Functionalities Unavailable in Solidity

One inline assembly fragment can implement several functionalities. For each functionality, we calculate the proportion of inline assembly fragments (shown in <>) implementing it. Moreover, to better understand each functionality, we added a real smart contract example to explain it. Note that these 12 functionalities cannot be realized by pure Solidity code.

F1<1.1%>: deploying a smart contract provided its bytecode using inline assembly operations `CREATE` or `CREATE2` [4]. Although a smart contract *A* can deploy a smart contract *B* using the keyword new of Solidity [3], there are two restrictions in Solidity. First, the source code of *B* is needed. Second, *B* should be available to *A* before compiling *A*. That is, *A* cannot deploy a smart contract whose source code or bytecode is given as input to *A*. These restrictions can be bypassed by inline assembly so that many smart contracts can be deployed by one transaction given their bytecode. Fig. 20 shows an example where deploy_SC() takes in the bytecode of *sc_num* (Line 2) smart contracts stored in *_code*. The bytecode of each smart contract in *_code* can be located by the variable *code_ptr* (Line 7). After `CREATE` is used to deploy a smart contract, its address is obtained (Line 9) and stored in the memory (Line 11). All addresses will be returned (Line 14).

Real contract: GAST2 Token [75] has 279,310 transactions, it uses the gas refund obtained when deleting the created contract so that the gas cost of executing the transaction is low [76]. And this contract uses **F1** to create a very simple contract to achieve this function.

F2<45.2%>: memory pointer, based on which smart contracts can manipulate the memory in a fine-grained way. **F2** can be realized by three inline assembly operations, `MLOAD` reading 32 bytes, `MSOTRE` writing 32 bytes and `MSTORE8` writing 8 bytes, which can access the memory location specified by developers [4].

Real contract: ENS [77] has 1,507,738 transactions, it implements a strlen function to obtain the length of the string. In the strlen() function, it uses `MLOAD` to get how many slots the string *s* occupies in the memory. Fig. 21 shows how the contract uses `MLOAD` to obtain the end pointer of string in memory (Line 5).

```
1  function deploy_SC(bytes[] _code) public returns (address[]) {
2      uint sc_num = _code.length;
3      address[] memory sc_addr = new address[](sc_num);
4      assembly {
5          for { let i := 0 } lt(i, sc_num) { i := add(i, 1) } {
6              //code_ptr points the i-th bytecode
7              let code_ptr := mload(add(_code, mul(0x20, add(i, 1))))
8              //deploy the bytecode, obtain its address
9              let target := create(0, add(code_ptr, 0x20), mload(code_ptr))
10             //record the address in sc_addr[]
11             mstore(add(sc_addr, mul(0x20, add(i, 1))), target)
12         }
13     }
14     return sc_addr;
15 }
```

Fig. 20. Deploy many smart contracts using one transaction

```
1  function strlen(string s) returns (uint) {
2      ...
3      assembly {
4          ptr := add(s, 1)
5          end := add(mload(s), ptr)
6      }
7      ...
8  }
```

Fig. 21. Obtain string length using mload

F3<8.2%>: obtaining the length of the allocated memory through the inline assembly operation `MSIZE` [4]. The memory slots after such length are unallocated [36], so knowing the length, inline assembly can avoid writing to the memory slots that are used by Solidity code.

Real contract: SyncFab [78] has 25,893 transactions, it implements the role of a proxy, which completes an external call and uses `MSIZE` to obtain the used memory size, and puts the returned result at the end to avoid overwriting other memory. Fig. 22 shows how the smart contract obtains the used memory size through `MSIZE` (Line 3) and stores the return data at the end (Line 5).

F4<0.7%>: storage pointer, based on which smart contracts can manipulate storage locations flexibly. **F4** can be realized

```
1  function _returnReturnData(bool _success) internal {
2      assembly {
3          let returndatastart := msize()
4          mstore(0x40, add(returndatastart, returndatasize))
5          returndatacopy(returndatastart, 0, returndatasize)
6          switch _success
7          case 0 { revert(returndatastart, returndatasize) }
8          default { return(returndatastart, returndatasize) }
9      }
10 }
```

Fig. 22. Obtain the used memory using msize

```
1  function _implementation() internal view returns (address impl) {
2      bytes32 slot = IMPLEMENTATION_SLOT;
3      assembly {
4          impl := sload(slot)
5      }
6  }
```

Fig. 23. Obtain the special stored bytes using sload

```
1 function () payable {
2    bytes32 result = _getAsset()._performGeneric...;
3    assembly {
4       mstore(0, result)
5       return(0, 32)
6    }
7 }
```

Fig. 24. Return data from a fallback function

```
 1 function _delegate(address implementation) internal {
 2    assembly {
 3       calldatacopy(0, 0, calldatasize)
 4       let result := delegatecall(gas, ...)
 5       returndatacopy(0, 0, returndatasize)
 6       switch result
 7       case 0 { revert(0, returndatasize) }
 8       default { return(0, returndatasize) }
 9    }
10 }
```

Fig. 25. Obtain the call result of the smart contract

by two inline assembly operations, SLOAD reading 32 bytes and SSTORE writing 32 bytes, which can access the storage location specified by developers [4]. Sophisticated functionalities can be built using storage pointers. For example, the diamond standard designs a new storage layout allowing to create structs in arbitrary places in the storage. Such layout mechanism relies on storage pointer [79]. A diamond is a contract with external functions that are supplied by contracts called facets [80]. Facets are separate, independent contracts that can share internal functions, libraries and state variables [80]. Diamonds can be upgraded without having to redeploy existing functionality. Parts of a diamond can be replaced while leaving other parts alone [80]. This contract architecture makes the upgrade of the contract more flexible, breaks the contract size limit, and maintains transparency.

Real contract: FiatTokenProxy [81] has 10,048,021 transactions, it is a proxy contract that uses a specific storage slot to store the address of a contract to be called. Its purpose is to update the address of the contract to be called by updating the specific storage of the contract. Fig. 23 shows how the smart contract obtains the bytes stored in the special storage through SLOAD and uses it for other operations (Line 4).

F5<9.3%>: returning data from a fallback function, which is an unnamed function in Ethereum smart contracts [35]. **F5** can be realized by inline assembly operations RETURN and REVERT, which return some data in the memory [4] and thus improve the capability of fallback functions.

Real contract: PolybiusToken [82] has 103,402 transactions, in order to prevent users from calling the wrong contract, it implements a proxy functionality in its fallback function, that is, if the user originally wants to call another function of another contract, its fallback function will be forwarded to the corresponding contract and use **F5** to return the call result in the fallback function. Fig. 24 shows how the smart contract returns the processing result in the fallback function through RETURN (Line 5).

F6<5.7%>: obtaining the return data from the previous invocation of a smart contract. Solidity does not support **F6** until V0.5.0 [83]. Before V0.5.0, **F6** must be implemented by an inline assembly operation RETURNDATACOPY [4].

```
1 function isContract(address _addr) returns (bool) {
2    if (_addr == 0) return false;
3    uint256 size;
4    assembly {
5       size := extcodesize(_addr)
6    }
7    return (size > 0);
8 }
```

Fig. 26. Determine whether the address is a smart contract address

Real contract: A proxy contract named FiatTokenProxy [81] has 10,048,021 transactions. Since this contract requires the return result of the contract it calls and the compiler version of it is below V0.5.0, RETURNDATACOPY is needed to obtain the returned data. Fig. 25 shows how the smart contract obtains the return date of the previous call to other smart contracts through RETURNDATACOPY (Line 5) and returns the returned data (Line 7).

F7<28.6%>: obtaining the runtime bytecode size of a given account through the inline assembly operation EXTCODESIZE [4]. Since an EOA does not contain bytecode, EXTCODESIZE can be used to check the type of an account. Moreover, the bytecode size is a mandatory parameter for copying the bytecode to the memory (i.e., **F8**) [4].

Real contract: SNT [84] has 954,549 transactions, it is a contract for the Status IM [85]. It is used by another control contract TokenController. Therefore, it is necessary for SNT to determine whether the sending address is a contract. This contract uses **F7** to obtain the runtime bytecode size of the address. If it is greater than zero, the address is a contract address, and then different logic will be made according to whether it is a contract address. Fig. 26 shows how the smart contract obtains the length of the runtime bytecode of the address through EXTCODESIZE (Line 5) and uses the length to determine whether the address is the address of the smart contract (Line 7).

F8<0.4%>: copying the runtime bytecode of a given account to a specified memory location. If the account is an EOA, nothing will be copied. The inline assembly operation, EXTCODECOPY can implement **F8** [4]. With **F8**, a smart contract can validate the bytecode of any specified smart contract. Fig. 27 shows a smart contract checking whether the first 44 bytes of a specified smart contract is identical with a prelude, which is generated at Lines 2 and 3. The first 44 bytes of a smart contract is read at Line 9, and the comparison is conducted at Lines 11 - 13.

Real contract: EIP20Factory [86] has 16 transactions, it is used to create a contract that meets EIP20 [87] and provides a function for verifying whether a contract address meets EIP20. Therefore, this contract needs to use **F8** to obtain the runtime bytecode of the contract and directly match all bytecodes to verify whether it is an ERC20 Token.

F9<0.3%>: obtaining the runtime bytecode size of the running contract through the inline assembly operation CODESIZE [4]. The bytecode size is a mandatory parameter for copying the bytecode to the memory (i.e., **F10**) [4].

```
function _getPrelude(address vaultContract)...(bytes memory prelude) {
    prelude = abi.encodePacked(bytes22(0x6e2b...5773),
      vaultContract, bytes2(0xff5b));
}
function _verifyPrelude(address metamorphicContract, bytes memory prelude)...{
    bytes memory runtimeHeader;
    assembly { .....
        //read the first 44 bytes of a smart contract
        extcodecopy(metamorphicContract, add(runtimeHeader, 0x20), 0, 44)
    }
    require(keccak256(abi.encodePacked(prelude)) ==
      keccak256(abi.encodePacked(runtimeHeader)),
      "Deployed runtime code does not have the required prelude.");
}
```

Fig. 27. Validate the bytecode of a given smart contract

```
1 function safeMemoryCleaner() internal pure {
2    assembly {
3       let fmem := mload(0x40)
4       codecopy(fmem, codesize, sub(msize, fmem))
5    }
6 }
```

Fig. 28. Clear memory using codesize and codecopy

```
1 function grab(bytes32 reveal, bytes32 commit, ...) public {
2    ...
3    bytes32 ticket;
4    assembly {
5       ticket := chainid()
6    }
7    ticket = keccak256(abi.encode(ticket, ...));
8    ...
9 }
```

Fig. 29. Obtain current blockchain id

```
1 function breed(uint256[2] mother, ...) {
2    assembly {
3       ...
4       function shiftR(value, offset) -> result {
5          result := div(value, exp(2, offset))
6       }
7       ...
8    }
9 }
```

Fig. 30. Restrict function scope using inline assembly function

F10<0.3%>: copying the runtime bytecode of the running contract to a specified memory location through the inline assembly operation, CODECOPY [4]. With **F10**, a smart contract can self check the integrity of its bytecode.

Real contract: a smart contract called MobiusRandom [88] has 2,769 transactions and uses both CODESIZE (**F9**) and CODECOPY (**F10**). Specifically, *this.code[codesize:length]* is used to get a piece of empty data (i.e., data with a value of 0), and CODECOPY is used to clear a piece of memory by copying the empty data into memory. Fig. 28 shows how the smart contract obtains the length of its runtime bytecode (Line 4) through CODESIZE and uses CODECOPY to clear a piece of memory (Line 4).

F11<0.08%>: obtaining the chain id of the running blockchain through the inline assembly operation CHAINID [4]. Since different blockchains have different chain ids (e.g., the chain ids of Ethereum mainnet and RSK mainnet are 1 and 30, respectively [89]), they can be used to differentiate blockchains. Moreover, with **F11**, a smart contract can check whether a signed message aims to be delivered to the running blockchain [90].

Real contract: OrchidLottery [91] has 753 transactions, it is a lottery contract, which uses CHAINID to perform different operations depending on the chain where it is located. In fact, CHAINID is used as a variable (similar to BLOCKHASH, BLOCKNUMBER, TIMESTAMP, etc.) for the hash operation, so if the chain is changed, the final hash result will also be affected, which is to make the result as unmanageable as possible. Fig. 29 shows how the smart contract obtains its own running blockchain id (Line 3) through CHAINID and uses it for the hash operation to obtain a more complex ticket (Line 4).

F12<0.04%>: functions with a restricted scope. Each inline assembly fragment has its own scope and the function defined using inline assembly is visible only to the code in the same fragment [4]. On the contrary, the function defined in Solidity has a global scope, which is visible to all other Solidity code in the same smart contract.

Real contract: PepeBase [92] has 8,133 transactions. Since the shift operation is not supported in Byzantium, the Solidity part will be replaced by other operations such as division. Due to the uncertainty of optimization, this contract directly uses **F12** to complete shift operation. Fig. 30 shows how the smart contract uses inline assembly function to encapsulate functions and does not expose the w/o inline part (Line 4).

We investigated the use of these 12 functionalities by the top 10 most popular open-source smart contracts with inline assembly. As shown in Table 2, among the top 10 open-source smart contracts with inline assembly involving the most transactions, **F2** is used by 6 contracts, **F7** by 5 contracts, and **F5** by 3 contracts, which matches the previous statistical results that **F2**, **F7** and **F5** are the top three functionalities with the largest proportion.

5.2 Saving Gas by Inline Assembly

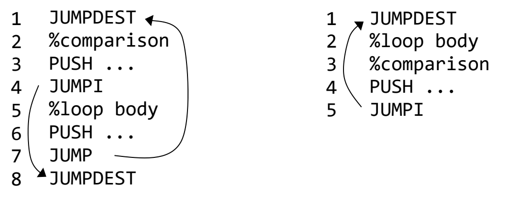
For each pattern, we count the number of inline assembly fragments implementing it, as shown in <>. One inline assembly can implement multiple patterns. Although a few tools such as Gasper [93], Syrup [94] and GasChecker [95] that can optimize smart contracts to save gas, but we believe inline assembly can complement these tools for two reasons. First, as the developers well know their smart contracts including coding and semantics, they can leverage inline assembly to directly optimize gas consumption. At the same time, they can use the optimizations provided by the Yul compiler. Second, other tools like Gasper, Syrup and GasChecker can only optimize the gas consumption of the Solidity source code or EVM bytecode that fulfills certain patterns, and thus they may miss some optimization opportunities known to the developers or the Yul compiler.

P1 $< 87 >$**:** do{...}while, can be realized by the inline assembly operation, JUMPI which is a conditional jump [4]. So-

Address	Name	Functionalities used
0x6090a6e47849629b7245-dfa1ca21d94cd15878ef	ENS	F2
0xece701c76bd00d1c3f96-410a0c69ea8dfcf5f34e	Etheroll	F2,F7
0x0affa06e7fbe5bc9a764-c979aa66e8256a631f02	PolybiusToken	F2,F5
0x86fa049857e0209aa7d9-e616f7eb3b3b78ecfdb0	DSToken	F2
0x744d70fdbe2ba4cf9513-1626614a1763df805b9e	SNT	F7
0x9899af5aa1efa90921d6-86212c87e70f4fbea035	Cryptaur	F2,F5
0xe7775a6e9bcf904eb39d-a2b68c5efb4f9360e08c	TAAS	F2,F5
0xdd98b423dc61a756e107-0de151b1485425505954	Dice	F7
0x55d34b686aa8c0492139-7c5807db9ecedba00a4c	StatusContribution	F7
0x960b236a07cf122663c4-303350609a66a7b288c0	MiniMeToken	F7

TABLE 2
Functionalities used in the top 10 most popular open-source smart contracts with inline assembly

lidity does not support do{...}while until V0.4.5 [96]. Before V0.4.5, Solidity has to use while{...} or for{...} to realize the functionality of do{...}while. However, while{...} and for{...} cost more gas than do{...}while because a loop guard will be evaluated before entering the loop for the first time. We use the EVM instructions compiled from a while{...} loop (Fig. 31(a)) and a do{...}while (Fig. 31(b)) to explain their differences in gas consumption in more detail.



```
1  JUMPDEST
2  %comparison
3  PUSH ...
4  JUMPI
5  %loop body
6  PUSH ...
7  JUMP
8  JUMPDEST
```

(a) A while/for{...} loop

```
1  JUMPDEST
2  %loop body
3  %comparison
4  PUSH ...
5  JUMPI
```

(b) A do{...}while loop

Fig. 31. EVM instructions of a loop with two implementations

In a while{...} loop (Fig. 31(a)), the program jumps over (Line 4) the loop body if the loop guard is evaluated to false (Line 2). Line 3 pushes the jump target on the stack. After executing the loop body (Line 5), the program jumps (Line 7) back to the start of the loop (Line 1). The jump target is pushed on the stack at Line 6. In a do{...}while loop (Fig. 31(b)), the loop guard is after the loop body (Line 3). If the loop guard is evaluated to true, the program jumps back to the start of the loop (Line 1). Line 4 pushes the jump target. We can see that a while{...} loop contains three more instructions (i.e., a `JUMPDEST`, a `PUSH` and a `JUMP`) than a do{...}while loop, so the deployment of a while{...} loop costs more gas. Assuming that the loop body executes n times, a while{...} loop will execute $2n$ more instructions (i.e., n `PUSH` and n `JUMP`) than a do{...}while loop. Therefore, the invocation of a while{...} loop costs more gas.

Six of the 87 smart contracts are compiled by compilers lower than V0.4.5, including a smart contract called Sweeper [97], which involves 35,370 transactions. This contract has only two functions with the same functionality, one of which is implemented by Solidity, and the other is implemented by inline assembly. And all transactions sent to the contract invoked the function implemented by inline assembly, which consumes less gas than the Solidity implementation. In addition, if the compiler version is not lower than V0.4.5, using the built-in do{...}while loop of Solidity can achieve the same gas-saving effect as using inline assembly. However, we observed that developers often use the for{...} loop and the while{...} loop instead of the do{...}while loop. Specifically, we checked all open-source smart contracts through string matching, and only found 141 contracts that used the do{...}while loop, while 3,597,421 and 150,718 contracts used the for{...} loop and the while{...} loop respectively.

```
1 function getSum(uint[] memory self)
2     public returns(uint) {
3     uint sum = 0;
4     for (uint i = 0; i < self.length; i++) {
5         sum += self[i];
6     }
7     return sum;
8 }
```

Fig. 32. Redundant bound checks at Line 5

In summary, we found that (1) the compiler before V0.4.5 does not support the do{...}while loop, and (2) developers rarely use the do{...}while loop, even though it is a gas-saving structure.

P2 $< 387 >$**:** redundant bound checks. The Solidity compiler generates a bound check for the access of an array item, if the item index is not a constant. However, the bound check is redundant if the index cannot overrun the array. Fig. 32 shows a practical smart contract whose gas consumption can be reduced if we replace Line 5 with inline assembly by eliminating the redundant bound checks for $self[i]$, whose index i is restricted by Line 4.

P3 $< 168 >$: type conversion. The type system of Solidity restricts the operations available to each type. Explicit type conversion is needed for a type to gain the ability of another type. For example, since an address is not allowed in arithmetic operations, the address should be converted to an integer before arithmetic operations. Type conversion will be compiled into EVM instructions, which cost gas. However, type conversion is not needed in inline assembly because inline assembly just supports one type, 256-bit unsigned integer [34].

P4 $< 109 >$**:** accessing an arbitrary-length substring of a string in the memory. The string in Solidity does not support access to its substring or individual characters [3]. To access a substring in Solidity, a smart contract must first convert the string into a *bytes* because the bytes type can be considered as a byte array that supports access to its individual bytes [3]. Then, a loop is needed to access the substring. By contrast, inline assembly can implement such functionality much simpler with less gas, because `MLOAD`, `MSTORE` and `MSTORE8` can access any specified memory slots [36]. Therefore, using inline assembly can avoid many array operations and the associated bound checks. Fig. 34(a) shows a smart contract that extracts a 32-byte substring starting from the 10th byte of a string. Line 3 converts the string into a byte. A loop (Lines 5, 6) is used for accessing the substring. Fig. 34(b) displays the inline assembly version. No loops, no array operations and no bound checks are used in inline assembly; instead, the inline assembly uses one `MLOAD` and one `MSTORE` to read and write the memory. We skip the first 32 bytes of *sub* (Line 4) and *str* (Line 5), because the bytes and the string are dynamically-sized types whose first 32 bytes store their lengths and their contents are stored immediately after the length fields [3].

P5 $< 113 >$**:** bitwise shift, can be realized by inline assembly operations `SHL`, `SHR` and `SAR` [4]. A bitwise shift operation in Solidity will be compiled into a multiplication instruction, `MUL` or a division instruction `DIV` before V0.5.5 [3]. Inline assembly saves gas because each of these three inline assembly operations costs 3 units of gas, but `MUL` and `DIV` cost 5 units of gas [36]. One of 113 is compiled with a version that is smaller than V0.5.5. After the V0.5.5 compiler, the Solidity implementation of bit operations will also be compiled into `SHL`, `SHR` and `SAR` instructions. That is, using inline assembly to implement bit operations does not save more gas than using Solidity. Nevertheless, 112 contracts still used inline assembly. A possible reason is that the update log of the compiler was not detailed enough so that the developers of these 112 contracts were unaware that Solidity consumes the same amount of gas as inline assembly.

P6 $< 1,635 >$**:** contract invocation. Inline assembly uses `CALL`, `CALLCODE`, `DELEGATECALL`, or `STATICCALL` to invoke another smart contract [4]. Each of them takes in many operands. For example, `CALL` takes in seven operands [4]. Therefore, the bytecode compiled from these inline assembly operations contains tens of instructions for preparing operands. Inline assembly saves gas compared to Solidity because Yul code has more optimization opportunities [34]. Fig. 33 shows the steps from source code to optimized bytecode. The compiler uses Yul Optimizer to optimize the inline assembly section. The other parts of the code are directly compiled into bytecode, so inline assembly will have additional optimizations. It is worth noting that after V0.7.5, the Solidity source code can be compiled into Yul code through a special compilation (i.e., –experimental-via-ir) option to enjoy the optimization of Yul [47]. However, the compilation option is turned off by default because Ethereum officials believe that this is a highly EXPERIMENTAL feature and cannot meet the product requirements [98].

To evaluate the impact of compilers on patterns, we conduct experiments on each pattern to explore whether they can save gas and how much gas they can save under different compiler versions. Specifically, for each pattern, we develop two smart contracts with the same functionality: one smart contract uses inline assembly whereas the other one is fully in Solidity. We compile them with different versions of compilers, then deploy and execute them in Remix to obtain gas consumption. The result shows that P1 and P5 can save gas under certain compiler versions, while other patterns can save gas under all current compiler versions (as shown in Table 3). Note that inline assembly is a feature only available in version 0.3.1 and later versions, and we chose the last sub-version of each version from 0.3 (released in August 2016) to 0.8 (released in November 2021) for this experiment.

Insight 5: Under specific compiler versions, developers can replace a block of gas-costly Solidity code with inline assembly to save gas according to the patterns above.

Version of Solc	0.3.6	0.4.26	0.5.17	0.6.12	0.7.6	0.8.10
P1	127	0	0	0	0	0
P2	200	235	205	205	205	205
P3	6	6	18	18	18	124
P4	10337	6529	6105	6105	6105	10617
P5	Unsupported	65	0	0	0	0
P6	21	36	84	84	84	84

TABLE 3
Gas saved by these patterns

F6, P1 and P5 are related to old compiler versions (e.g., P1 can save gas only under a certain compiler version or older versions). To understand the use of old compilers, we count the distribution of compiler versions used in the deployment of the smart contracts with inline assembly. Fig. 35 shows the top 9 compiler versions of all open-source contracts with inline assembly in the first three months of 2020. The most used compiler version is V0.5.4 (released in February 2019), accounting for 23.1%. The second most used is V0.4.24 (released in May 2018), accounting for 21.0%, the third to ninth are V0.4.23 (released in April 2018) accounting for 20.4%, V0.5.10 accounting for 16%, V0.5.15 accounting for 4.0%, V0.5.12 accounting for 3.7%, V0.5.2 accounting for 3.4%, V0.5.11 accounting for 3.4%, the rest of versions accounting for 4.8% respectively. The results show that the top three compiler versions used most in the open-source smart contract in 2020 were all released a year ago, which indicates that many open-source smart contracts are still compiled with old compilers. It also shows that our research on functionalities and gas-saved patterns of the old compilers are meaningful.

5.3 Easing Development through Inline Assembly

D1 $< 2,517 >$**:** switch...case is only supported by inline assembly [4]. For the same functionality, Solidity code has to use if...else with multiple branches, which is clumsy [99].

Answer to RQ3: Inline assembly is used for implementing unavailable functionalities in Solidity, optimizing bytecode to save gas, or ease of programming.

6 Questionnaire Survey

To understand the usage of the above 12 functionalities among smart contracts developers and their opinions on inline assembly, we designed and released this survey, and received informative feedback from 122 smart contract developers in 22 countries. The survey result can support our findings. For example, the result is accordant with the answer to RQ1 that inline assembly is frequently used and our insights/observations in §4 and §5, including (1) developers can use some specific opcodes directly through inline assembly to implement functionalities unavailable in Solidity; (2) developers tend to use inline assembly with short length to implement uncomplicated control flows in long smart contracts; (3) developers using inline assembly

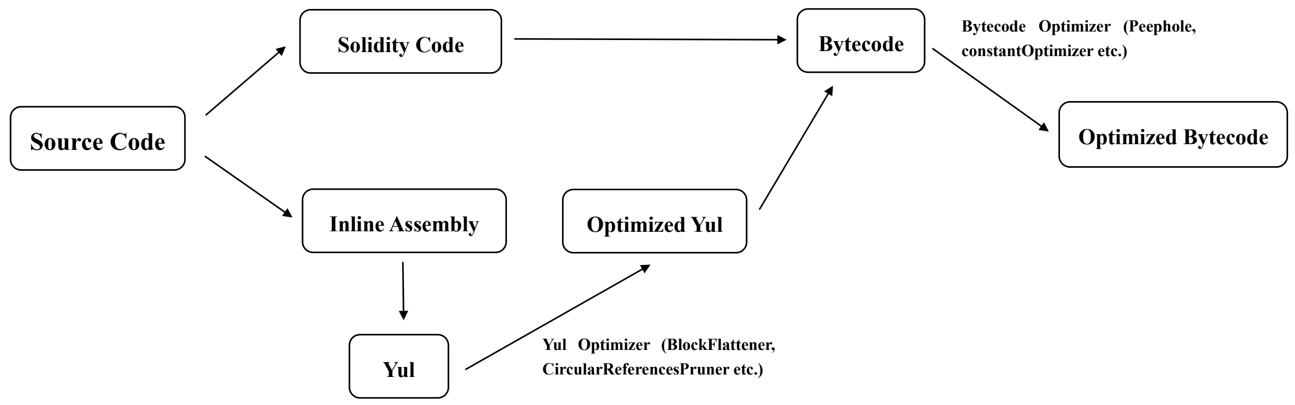

Source Code
Solidity Code
Inline Assembly
Yul
Optimized Yul
Yul Optimizer (BlockFlattener, CircularReferencesPruner etc.)
Bytecode
Bytecode Optimizer (Peephole, constantOptimizer etc.)
Optimized Bytecode

Fig. 33. The process of bytecode optimization.

```
function substr(string str)
  returns (bytes sub) {
    bytes memory str_bytes =  bytes(str);
    sub = new bytes(32);
    for (uint i = 10; i < 42; i++) {
        sub[i-10] = str_bytes[i];
    }
}
```

(a) Source code version

```
function substr(string str)
  returns (bytes sub) {
    assembly {
        mstore(add(sub, 0x20),
          mload(add(str, add(0x20, 10)))
    }
}
```

(b) Inline assembly version

Fig. 34. Extract a 32-byte substring starting from the 10th byte of a string

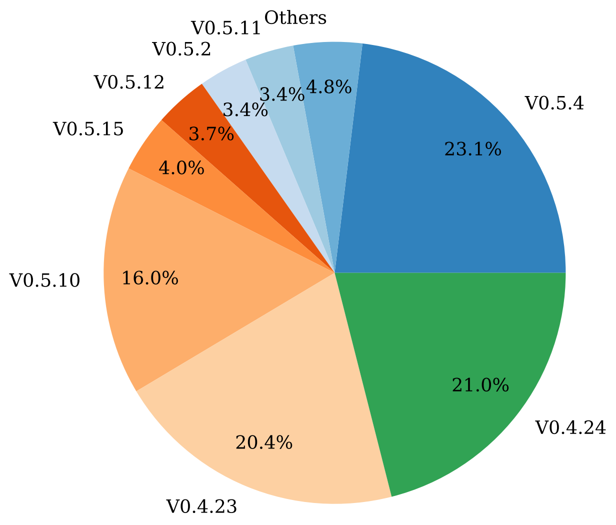


Fig. 35. Compiler versions of contracts with inline assembly

may pay more attention to the performance of smart contracts than those writing pure Solidity code. More precisely, 57 (46.7%) respondents use inline assembly to implement some functionalities unavailable in Solidity; 43 (35.2%) use inline assembly to save gas; 21 (17.2%) use inline assembly because it is more concise than Solidity code. Moreover, the results of this survey also point out some suggestions for improving smart contract IDE (Integrated Development Environment) and Solidity compilers, such as the need of tools for supporting the development of inline assembly and checking its accuracy and security, the support of those 12 functionalities that can only be implemented in inline assembly.

6.1 Survey design

According to the instructions of Kitchenham et al. for personal opinion surveys [100], we conduct an anonymous questionnaire survey to increase the response rates[1] We first conduct a small-scale survey to test and refine our questions based on feedbacks, e.g., "the question is not clear". After that, we conduct a larger-scale survey sending to 1,258 practitioners and received 122 responses finally. Our survey consists of 23 questions. The first seven questions concern the characteristics of participants, e.g., what is the resident country, how many years of experience the participant has about smart contracts, what is the main role in developing smart contracts, do you think whether the participant uses inline assembly of smart contracts? The next three questions focus on the reasons for using inline assembly, the difficulty of using inline assembly and Solidity, and the verification of whether to choose inline assembly if it consumes less gas than Solidity. The 11th–22nd questions list the 12 functionalities that can only be realized by inline assembly (§5.1). For each functionality, we ask the participants whether the functionality is useful with three options, *Very*, *Sometimes* and *No* and whether participants want Solidity to support this functionality with three options, *Yes*, *Neutral* and *No*. We also add an option that allows participants to leave comments for each functionality. The final question allows participants to write down any other comments. All questions listed in the survey are presented in the supplementary material [43].

1. Our IRB application for conducting this questionnaire survey has been approved by The Hong Kong Polytechnic University.

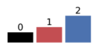
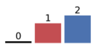
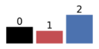
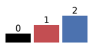
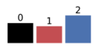
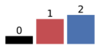
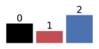
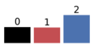
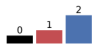
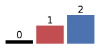
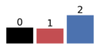
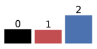
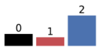
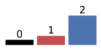
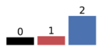
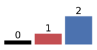
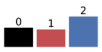
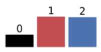
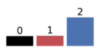
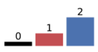
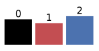
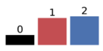
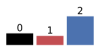
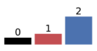
Fun	Useful?		Add in Solidity?		Fun	Useful?		Add in Solidity?	
	Distribution	Score	Distribution	Score		Distribution	Score	Distribution	Score
1		1.3		1.5	7		1.2		1.3
2		1.1		1.3	8		1.1		1.2
3		1.4		1.5	9		1.2		1.2
4		1.4		1.5	10		1.5		1.5
5		1.2		1.2	11		1.4		1.5
6		1.0		1.3	12		1.3		1.5

TABLE 4
Scores of questions about 12 functionalities in §5.1

6.2 Results

We received 122 responses (the response rate is 9.7%) from 22 different countries, and received 46 comments. The average years of experience in smart contracts are 2.02 years, indicating good experiences of respondents since Ethereum was launched in 2015. 63.9% of participants are developers of smart contracts, and impressively, 49 (40%) participants say they have *ever* used inline assembly in developing smart contracts. The result is accordant with the answer to RQ1 that inline assembly is frequently used. 57 (46.7%) respondents use inline assembly to implement some functionalities unavailable in Solidity; 43 (35.2%) use inline assembly to save gas; 21 (17.2%) use inline assembly because it is more concise than Solidity code. 77 (64%) participants believe that inline assembly is more difficult to write than Solidity. If a functionality can be implemented by both inline assembly and Solidity but the inline assembly version saves gas, 63 participants (51.6%) would choose inline assembly version.

Table 4 shows the results of questions about the unavailable functionalities in Solidity. Columns 1 and 6 present the functionalities. We give the options, *Very*, *Sometimes* and *No* scores 2, 1, 0 to quantify the functionality usefulness. Similarly, we give the options, *Yes*, *Neutral* and *No* scores 2, 1, 0 to quantify the willingness of participants to have the functionality in Solidity. Columns 2, 4, 7, 9 illustrate the score distributions, and columns 3, 5, 8, 10 present the average scores. All average scores of functionality usefulness are no lower than 1.0, indicating that participants believe inline assembly can complement Solidity. Moreover, all average scores of the willingness to build these functionalities in Solidity are no lower than 1.2, implying that participants expect to realize these functionalities in Solidity in future.

Insight 6: Practitioners believe inline assembly is useful, and show positive attitudes to add these functionalities in Solidity.

We then investigate why 9 participants believe these functionalities are not useful and 4 participants do not want Solidity to add these functionalities by reading their comments. 6 believe some functionalities (e.g., **F2**) have high security risks, 4 consider some functionalities (e.g., **F11**) are useful in very limited applications, 2 have never used some functionalities (e.g., **F3**) in their smart contracts, and 1 worries about the maintenance difficulty of inline assembly.

We receive many positive comments about our survey, including:
– Good questions you've listed. You should publish these in a post, they're interesting parts of Solidity/EVM;
– Very nice survey, keep up the good work and please send me the results;
– These were good ideas for improving Solidity.

7 Threats to Validity

Internal validity. The inserted marks may change the instructions in the original smart contract, or the original smart contract may contain the same instructions as the inserted marks, which may affect the accuracy of bytecode segment extraction. To reduce this threat, we propose three requirements for producing marks, and we correlate all 19,073 and 1,824 unique extracted bytecode segments with its inline assembly fragment manually to ensure correctness. Specifically, we disassemble the extracted bytecode of all inline assembly fragments, then compare the instructions obtained by disassembly with the source code of inline assembly fragment to determine whether the bytecode of the inline assembly fragment we extracted is complete. That is, the extracted bytecode is complete only if the semantics of the instructions obtained by disassembly is the same as the source code of the inline assembly fragment. We also use regular matching to solve the problem that the bytecode of end mark cannot be generated because of early return/revert in inline assembly, that is, match specific instructions (i.e., `RETURN`, `REVERT`) and treat them as the end of inline assembly.

We find that estimateGas() uses a binary search method to estimate gas consumption, which leads to overestimation [101]. To reduce this threat, we deployed smart contracts in Remix [56] and executed real transactions to measure gas consumption. Specifically, for each pattern, we first developed a smart contract with inline assembly and another smart contract fully in Solidity with the same functionality. Secondly, we compiled the two smart contracts in Remix using the same compiler and compilation options, and then deployed them to virtual machine of Remix. Finally, we called these two smart contracts with the same parameters and obtained the consumed gas. Since the smart contracts we developed do not depend on the blockchain state (e.g., the block number) and the smart contract state (e.g., attribute variables), Remix returned constant gas consumption values. We acknowledged that Remix may return infinite gas consumption in some cases, for example, the executed function reads the blockhash of a specific block [102]. Since our patterns are not related to the cases incurring infinite estimated gas consumption, we did not encounter infinite gas consumption in our empirical study.

For the threat relevant to the implementation of inline assembly, we ask multiple developers to write Solidity for implementing the same functionality and then conduct the analysis. Besides, this empirical study involves much manual effort, which depends on researcher's experience and ability significantly. To reduce this threat, we ask four students to analyze independently, and then resolve their inconsistencies by the discussion of two authors with more than four years of experience in developing smart contracts.

Moreover, participants may not have good understandings of our survey. To reduce this threat, we allow participants to leave comments. We revise our survey if some participants have problems in understanding questions.

External validity. Solidity and Yul are fast-evolving languages, and new features have been frequently introduced, which may pose new threats to inline assembly. To reduce this threat, we analyze all open-source smart contracts deployed before March 2020, and we adopt the corresponding version of compiler that was used to compiled each smart contract to ensure that the generated bytecode is the same as the bytecode deployed on the blockchain. The methodology of our study is also applicable to new language features and new smart contracts. Due to the need for source code support, this study only discusses and analyzes open-source smart contracts, and can only explain the current inline assembly usage of open-source smart contracts. Because in the compiled bytecode, the inline assembly has no specific start and end marks, so it is very difficult to analyze whether the bytecode of the smart contract contains the inline assembly or which part is the bytecode generated by the inline assembly without the source code. Therefore, all observations and conclusions of this paper are derived from the analysis of open-source smart contracts.

8 Related Work

Vulnerability discovery. Nguyen et al. present an adaptive fuzzer for Ethereum smart contracts called sFuzz, which combines the strategy in the AFL fuzzer and an efficient lightweight multi-objective adaptive strategy to target branches that are difficult to cover [103]. ETHBmc, a bounded model checker based on symbolic execution, can be used to detect suicidal and unsecured balance vulnerabilities [104]. sCompile statically identifies the paths involving monetary transactions in smart contracts. It can detect four types of security vulnerabilities [105]. WANA is a cross-platform smart contract vulnerability detection tool based on the symbolic execution of WebAssembly bytecode. It can detect three typical vulnerabilities for Ethereum smart contracts [106]. EOSFuzzer, a general black-box fuzzing framework, is used to detect vulnerabilities within EOSIO smart contracts [107]. Ashraf et al. present GasFuzzer, a tool that fuzzes Ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities [108]. Harvey is a greybox fuzzer for smart contracts that selects appropriate inputs and transaction sequences to increase code coverage [109]. ContractGuard inserts checks into smart contracts to protect them from intrusion attacks [110]. VULTRON compares the actual transferred amount and the amount reflected on the contract's internal bookkeeping to detect malicious transactions [111]. Chen et al. define 20 smart contract defects on Ethereum by analyzing real-world smart contracts and the posts on StackExchange [112]. SoCRATES uses a federated organization of bots to mimic the interactions among users to expose known and unknown faults in smart contracts [113]. Chen proposes a deep learning-based method to find security issues of Ethereum smart contracts by finding the updated version of a destructed contract [114]. SOLC-VERIFY is a tool for automated verification of Solidity smart contracts based on modular program reasoning and SMT solvers [26]. Its author stated that the tool does not support inline assembly but did not give a specific reason. Similarly, a safety verifier for Ethereum smart contracts called VERISMART does not support inline assembly, and the authors admit that the tool may miss bugs hidden in inline assembly [27]. In summary, these studies focus on security vulnerabilities rather than inline assembly in smart contracts.

Semantics. Jiao et al. develop the operational formal semantics for Solidity which defines correct and secure high-level execution behaviors of smart contracts to reason about compiler bugs and assist developers in writing secure smart contracts [28], [29]. However, they clarify that their work skips the tests for inline assembly and the proposed Solidity semantics do not cover inline assembly statements because they believe that inline assembly is a low-level feature to access EVM [28], [29]. Zakrzewski proposes a formalization of a small subset of Solidity that contains its core data model and some unique features [30]. And he mentioned that inline assembly is incompatible with their high-level modeling and cannot be implemented [30].

Gas. sOptimize identifies and removes three kinds of code blocks to optimize smart contract gas consumption [115]. Khan et al. analyze the factors causing an increase or decrease in the gas consumption of smart contracts [116]. Zarir et al. analyze the gas usage of Ethereum transactions and obtain some new observations and insights [117]. Marchesi et al. provide a set of design patterns and tips to help gas saving in developing smart contracts [118]. Hukkinen et al. pinpoint inefficiencies in the design of the smart contract and reduce its gas consumption, and then formulate a set of guidelines suitable for optimizing the efficiency of smart contracts [119]. Chen et al. propose and develop three tools named GASPER [93], GasChecker [95] and GasReducer [120]. GASPER can automatically locate gas-costly patterns in smart contracts [93], and GasChecker is an extended version of the former [95]. And GasReducer can detect anti-patterns and replace them with efficient code [120]. Wang et al. reduce the cost of gas by automatically generating cost-effective and representative test suites [121]. Visualgas is a tool for in-depth visualization of gas costs to support the simpler and more gas-efficient development of smart contracts [31]. It does not support the use of inline assembly in Solidity because the author believes that inline assembly can break many invariants that the solc compiler would comply with [31].

In addition, there are some other studies on gas, including estimation of gas consumption [53], [122], [123], [124], [125], [126], [127], gas price [128], [129], [130], [131], [132], [133], gas mechanism [134], [135], [136], the out of gas exceptions [137], [138] and others [139]. However, none of the above work uses inline assembly for gas optimization or other purposes.

Performance. Zheng et al. consider gas as an unsuitable metric by its one-sidedness and floating [140]. They propose detailed performance metrics and a performance monitoring framework with a log-based method to provide performance monitoring approach and real-time performance information for different blockchain systems [140]. However, the inline assembly is not mentioned in this work.

Development and maintenance. Zou et al. reveal several major challenges developers are facing during smart contract development through interviews and surveys [33]. Zheng et al. present the challenges in smart contracts as well as recent technical advances [141]. Khan et al. present a taxonomy of existing blockchain-enabled smart contract solutions [142]. Chen et al. conduct an empirical study of code reuse in smart contracts [143]. Zhang et al. propose a source code obfuscation approach for Ethereum smart contracts [144].

There are also many empirical studies and surveys about smart contracts and blockchains [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24], but to the best of our knowledge, none of them investigate the usage of inline assembly in smart contracts.

Inline assembly in C programs. Although a recent work studied the x86-x64 inline assembly in C programs [32], e.g., how common is inline assembly in C programs, why C programs use inline assembly, its methodology and insights cannot be directly used in our study due to the fundamental differences between smart contracts and C programs running on x86/x64 CPUs. More precisely, Ethereum has a new instruction set with more than 130 instructions [36] rather than x86/x64. Moreover, the inline assembly of Solidity and Yul have different syntax and semantics from C and x86/x64 inline assembly [3]. Smart contracts are interpreted by the stack-based EVM [35] instead of CPU. Besides, Ethereum introduces a *gas* mechanism which affects smart contract development, deployment and execution [93], [120]. The x86/x64 CPU does no have the gas mechanism.

Besides the differences between smart contracts and C programs, our methodology has three obvious differences with the related work [32]. First, we collect open-source smart contracts and transactions by leveraging Etherscan and BigQuery, while the C programs studied in [32] were fetched from Github. It is worth noting that, although there has been related work [48] has collected all the open-source smart contracts and transactions, but the open-source content only has the source code without the compiled version and whether it is optimized of the smart contract required by this paper. Second, we investigate both static and dynamic features of smart contracts, whereas only static features were studied in [32]. Third, our study needs to address Ethereum-specific challenges (e.g., extracting inline assembly bytecode segments) and includes some unique evaluation (e.g., gas consumption) and comparison (e.g., comparing smart contracts with and smart contracts without inline assembly) that are not involved in [32].

9 Conclusion and Future Work

We conduct the first large-scale empirical study of inline assembly on Ethereum smart contracts. We obtain new insights and reveal why smart contracts use inline assembly by a thorough quantitative and qualitative analysis, and a survey. This study can help developers in the following three points. First, this study can provide suggestions to developers when they can use inline assembly. Second, this study can provide suggestions to developers on how to use inline assembly to save gas under specific compiler versions. Third, this study can also benefit the evolution of Solidity and the optimization of its compilers. For example, in the survey, many developers hope that certain functionalities of inline assembly can be implemented in future versions of Solidity.

In future work, we will explore three directions. First, we will develop a tool to provide developers suggestions on how to optimize their smart contracts by replacing codes in Solidity with inline assembly. Second, we will consider implementing a bytecode segment extraction tool that developers and researchers can leverage. Our bytecode segment extraction tool can not only support the positioning of Yul inline assembly, but also the positioning of specified statements in solidity. Third, Solidity and Yul are fast-evolving languages, and tens of thousands of smart contracts are deployed everyday, which means that existed language features may be deleted, new features may be introduced and new usages of inline assembly may appear in future. We will continue to track the new features of the compiler and analyze whether these new features pose any threats to inline assembly.

Acknowledgement

This work was partly supported by National Natural Science Foundation of China (61872057), National Key RD Program of China (2018YFB0804100), Hong Kong RGC Projects (PolyU15222320, PolyU15224121), Science and Technology Development Fund of Macau under Grant 0047/2020/A1, China Postdoctoral Science Foundation under Grants 2017M621247, and Natural Science Foundation of Heilongjiang Province, China under Grant LH2019F008.

References

[1] I. Koksal. (2019) The benefits of applying blockchain technology in any industry. [Online]. Available: https://www.forbes.com/sites/ilkerkoksal/2019/10/23/the-benefits-of-applying-blockchain-technology-in-any-industry/#6da7d2b949a5

[2] Google. (2020) Big query. [Online]. Available: https://console.cloud.google.com/bigquery?project=bigquery-public-data

[3] Ethereum. (2020) Solidity documentation. [Online]. Available: https://solidity.readthedocs.io/en/latest/

[4] Ethereum. (2020) Inline assembly. [Online]. Available: https://solidity.readthedocs.io/en/latest/assembly.html

[5] Y. Chen and C. Bellavitis, "Blockchain disruption and decentralized finance: The rise of decentralized business models," *Journal of Business Venturing Insights*, vol. 13, p. e00151, 2020.

[6] DefiLlama. (2021) Total value locked rankings. [Online]. Available: https://defillama.com/chain/Ethereum

[7] Stack Exchange. (2020) Ethereum stack exchange. [Online]. Available: https://ethereum.stackexchange.com/

[8] Reddit. (2020) Reddit: the front page of the internet. [Online]. Available: https://www.reddit.com

[9] gensim. (2020) models.ldamodel — latent dirichlet allocation. [Online]. Available: https://radimrehurek.com/gensim/models/ldamodel.html

[10] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *the Journal of machine Learning research*, vol. 3, pp. 993–1022, 2003.

[11] K. Wu, Y. Ma, G. Huang, and X. Liu, "A first look at blockchain-based decentralized applications," *Software: Practice and Experience*, 2019.

[12] Z. Wan, X. Xia, and A. E. Hassan, "What is discussed about blockchain? a case study on the use of balanced lda and the reference architecture of a domain to capture online discussions about blockchain platforms across the stack exchange communities," *IEEE Transactions on Software Engineering*, 2019.

[13] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug characteristics in blockchain systems: a large-scale empirical study," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 413–424.

[14] A. Pinna, S. Ibba, G. Baralla, R. Tonelli, and M. Marchesi, "A massive analysis of ethereum smart contracts empirical study and code metrics," *IEEE Access*, vol. 7, pp. 78 194–78 213, 2019.

[15] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns," in *International conference on financial cryptography and data security*. Springer, 2017, pp. 494–509.

[16] M. Coblenz, J. Aldrich, J. Sunshine, and B. Myers, "An empirical study of ownership, typestate, and assets in the obsidian smart contract language," *arXiv preprint arXiv:2003.12209*, 2020.

[17] V. Saraph and M. Herlihy, "An empirical study of speculative concurrency in ethereum smart contracts," *arXiv preprint arXiv:1901.01376*, 2019.

[18] G. A. Oliva, A. E. Hassan, and Z. M. J. Jiang, "An exploratory study of smart contracts in the ethereum blockchain platform," *Empirical Software Engineering*, pp. 1–41, 2020.

[19] I. Mokdad and N. M. Hewahi, "Empirical evaluation of blockchain smart contracts," in *Decentralised Internet of Things*. Springer, 2020, pp. 45–71.

[20] M. Möhring, B. Keller, R. Schmidt, A.-L. Rippin, J. Schulz, and K. Brückner, "Empirical insights in the current development of smart contracts." in *PACIS*, 2018, p. 146.

[21] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," 2020.

[22] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," *arXiv preprint arXiv:1809.02702*, 2018.

[23] R. M. Parizi, A. Dehghantanha *et al.*, "Smart contract programming languages on blockchains: an empirical evaluation of usability and security," in *International Conference on Blockchain*. Springer, 2018, pp. 75–91.

[24] 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 *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–11.

[25] T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang, "Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum," in *In ACM Conference on Computer and Communications Security (CCS)*, 2019.

[26] Á. Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," *arXiv preprint arXiv:1907.04262*, 2019.

[27] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "Verismart: A highly precise safety verifier for ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1678–1694.

[28] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of solidity," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1695–1712.

[29] J. Jiao, S.-W. Lin, and J. Sun, "A generalized formal semantic framework for smart contracts." in *FASE*, 2020, pp. 75–96.

[30] J. Zakrzewski, "Towards verification of ethereum smart contracts: a formalization of core of solidity," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2018, pp. 229–247.

[31] C. Signer, "Gas cost analysis for ethereum smart contracts," Master's thesis, ETH Zurich, Department of Computer Science, 2018.

[32] M. Rigger, S. Marr, S. Kell, D. Leopoldseder, and H. Mössenböck, "An analysis of x86-64 inline assembly in C programs," in *In Proc. ACM International Conference on Virtual Execution Environments*, 2018.

[33] W. Zou, D. Lo, P. S. Kochhar, X.-B. D. Le, X. Xia, Y. Feng, Z. Chen, and B. Xu, "Smart contract development: Challenges and opportunities," *IEEE Transactions on Software Engineering*, 2019.

[34] Ethereum. (2020) Yul. [Online]. Available: https://solidity.readthedocs.io/en/latest/yul.html

[35] Ethereum. (2020) Ethereum Homestead documentation. [Online]. Available: https://ethdocs.org/en/latest/

[36] G. Wood. (2020) Ethereum: A secure decentralised generalised transaction ledger. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[37] Ethereum. (2020) The optimiser. [Online]. Available: https://solidity.readthedocs.io/en/latest/internals/optimiser.html

[38] Ethereum. (2020) Ethereum blockchain explorer. [Online]. Available: https://etherscan.io/contractsVerified

[39] A. Day and E. Medvedev. (2018) Ethereum in BigQuery: a public dataset for smart contract analytics. [Online]. Available: https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics

[40] (2020) Api used for collecting the source code of smart contracts. [Online]. Available: https://api.etherscan.io/api?module=contract&action=getsourcecode&address=TheContractAddress&apikey=YourApiKeyToken

[41] Etherscan. (2020) Ethereum daily verified contracts chart. [Online]. Available: https://etherscan.io/chart/verified-contracts

[42] Ethereum. (2020) solidity documents. [Online]. Available: https://docs.soliditylang.org/en/v0.8.6/internals/optimizer.html

[43] Z. Liao, S. Song, H. Zhu, X. Luo, Z. He, R. Jiang, T. Chen, J. Chen, T. Zhang, and X. Zhang, "Large-scale empirical study of inline assembly on 7.6 million ethereum smart contracts (supplementary material)."

[44] Enzyme Finance. (2021) Oyente – an analysis tool for smart contracts. [Online]. Available: https://github.com/enzymefinance/oyente

[45] Neville Grech. (2021) Madmax – ethereum static vulnerability detector for gas-focussed vulnerabilities. [Online]. Available: https://github.com/nevillegrech/MadMax

[46] Harry Altman. (2021) Yul sourcemaps skip let statements with no initializer. [Online]. Available: https://github.com/ethereum/solidity/issues/8838

[47] Ethereum. (2015) Solidity, the smart contract programming language. [Online]. Available: https://github.com/ethereum/solidity/blob/develop/Changelog.md

[48] P. Zheng, Z. Zheng, and H. ning Dai, "XBlock-ETH: Extracting and exploring blockchain data from Ethereum," *IEEE Open Journal of the Computer Society*, 2020.

[49] H. Kalodner, S. Goldfeder, A. Chator, M. Möser, and A. Narayanan, "Blocksci: Design and applications of a blockchain analysis platform," in *USENIX Security Symposium*, 2020.

[50] J. D. Gibbons and S. Chakraborti, *Nonparametric statistical inference*. CRC press, 2020.

[51] Wikipedia. (2020) Mann–Whitney U test. [Online]. Available: https://en.wikipedia.org/wiki/Mann-Whitney_U_test

[52] S. Nystedt, C. Sandros *et al.*, "Software complexity and project performance," *School of Economics and Commercial Law at the University of Gothenburg*, 1999.

[53] P. Hegedus, "Towards analyzing the complexity landscape of solidity based ethereum smart contracts," *Technologies*, vol. 7, no. 1, 2019.

[54] Etherscan. (2021) Contract – 0x1e143b2588705DfEA63A17f2032CA123dF995CE0 . [Online]. Available: https://etherscan.io/address/0x1e143b2588705dfea63a17f2032ca123df995ce0#code

[55] Etherscan. (2021) Contract – 0x20A7b20b9c213E6705c72a4216Fdbc628A29d06C . [Online]. Available: https://etherscan.io/address/0x20a7b20b9c213e6705c72a4216fdbc628a29d06c#code

[56] Ethereun. (2018) Remix - Ethereum Browser-based compiler and ide. [Online]. Available: https://remix.ethereum.org/

[57] Etherscan. (2021) Contract – 0x185479FB2cAEcbA11227db4186046496D6230243 . [Online]. Available: https://etherscan.io/address/0x185479fb2caecba11227db4186046496d6230243#code

[58] Etherscan. (2021) Contract – 0x9bCd203710b2382CE896c5dAd7342c3dc444544E . [Online]. Available: https://etherscan.io/address/0x9bcd203710b2382ce896c5dad7342c3dc444544e#code

[59] Etherscan. (2021) Contract – 0x2B9B8b83C09e1d2BD3DacE1C3dB2fEc8ff54a8AC . [Online]. Available: https://etherscan.io/address/0x2b9b8b83c09e1d2bd3dace1c3db2fec8ff54a8ac#code

[60] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, and I. Naseer, "Formal specification and verification of smart contracts for azure blockchain," *arXiv preprint arXiv:1812.08829*, 2018.

[61] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[62] P. Zhang, F. Xiao, and X. Luo, "Soliditycheck: Quickly detecting smart contract problems through regular expressions," *arXiv preprint arXiv:1911.09425*, 2019.

[63] Etherscan. (2020) Contract – 0x84F6451efE944ba67BedB8E0Cf996Fa1fEB4031D. [Online]. Available: https://etherscan.io/address/0x84f6451efe944ba67bedb8e0cf996fa1feb4031d#code

[64] Etherscan. (2020) Contract – 0xD01c92937400DD1ecE24992B1dc44Aeaa47Ae72a. [Online]. Available: https://etherscan.io/address/0xd01c92937400dd1ece24992b1dc44aeaa47ae72a#code

[65] Etherscan. (2020) Contract – 0x0FbdB0aCEb333e99f68b700Ed5ffE824Bb9205ee. [Online]. Available: https://etherscan.io/address/0x0fbdb0aceb333e99f68b700ed5ffe824bb9205ee#code

[66] Wikipedia contributors. (2021) Zero-knowledge proof — Wikipedia, the free encyclopedia. [Online]. Available: https://en.wikipedia.org/wiki/Zero-knowledge_proof

[67] Loopring. (2017) Protocol – a zkrollup dex & payment protocol. [Online]. Available: https://github.com/Loopring/protocols

[68] T. Chen, Z. Li, Y. Zhu, J. Chen, X. Luo, J. C.-S. Lui, X. Lin, and X. Zhang, "Understanding ethereum via graph analysis," *ACM Transactions on Internet Technology (TOIT)*, vol. 20, no. 2, pp. 1–32, 2020.

[69] Etherscan. (2021) Contract – 0x3B9A3c062Bdb640b5039C0cCda4157737d732F95. [Online]. Available: https://etherscan.io/address/0x3b9a3c062bdb640b5039c0ccda4157737d732f95#code

[70] Etherscan. (2021) Contract – 0xBC0BB74a13f1455E9eCCf3275bC96855A3c5BA7B. [Online]. Available: https://etherscan.io/address/0xbc0bb74a13f1455e9eccf3275bc96855a3c5ba7b#code

[71] Etherscan. (2021) Contract – 0x5cf3da4D17b83b07a73fE6b706E9D394113052cf. [Online]. Available: https://etherscan.io/address/0x5cf3da4d17b83b07a73fe6b706e9d394113052cf#code

[72] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhange, "Understanding Ethereum via graph analysis," in *in Proc. IEEE conference on computer communications*, 2018.

[73] Wikipedia contributors. (2021) Cyclomatic complexity — Wikipedia, the free encyclopedia. [Online; accessed 17-July-2021]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Cyclomatic_complexity&oldid=1016694231

[74] Wikipedia. (2021) Phi coefficient. [Online]. Available: https://en.wikipedia.org/wiki/Phi_coefficient

[75] Etherscan. (2021) Contract – 0x0000000000b3F879cb30FE243b4Dfee438691c04. [Online]. Available: https://etherscan.io/address/0x0000000000b3f879cb30fe243b4dfee438691c04#code

[76] GasToken. (2019) Tokenize gas on ethereum with gastoken. [Online]. Available: https://gastoken.io/

[77] Etherscan. (2021) Contract – 0x6090A6e47849629b7245Dfa1Ca21D94cd15878Ef. [Online]. Available: https://etherscan.io/address/0x6090a6e47849629b7245dfa1ca21d94cd15878ef#code

[78] Etherscan. (2021) Contract – 0x6710c63432A2De02954fc0f851db07146a6c0312. [Online]. Available: https://etherscan.io/address/0x6710c63432a2de02954fc0f851db07146a6c0312#code

[79] N. Mudge. (2020) New storage layout for proxy contracts and diamonds. [Online]. Available: https://medium.com/1milliondevs/new-storage-layout-for-proxy-contracts-and-diamonds-98d01d0eadb

[80] Ethereum. (2020) Eip-2535: Diamonds. [Online]. Available: https://eips.ethereum.org/EIPS/eip-2535

[81] Etherscan. (2021) Contract – 0xA0b86991c6218b36c1d19D4a2e9Eb0cE3606eB48. [Online]. Available: https://etherscan.io/address/0xa0b86991c6218b36c1d19d4a2e9eb0ce3606eb48#code

[82] Etherscan. (2021) Contract – 0x0AfFa06e7Fbe5bC9a764C979aA66E8256A631f02. [Online]. Available: https://etherscan.io/address/0x0affa06e7fbe5bc9a764c979aa66e8256a631f02#code

[83] Ethereum. (2018) Solidity v0.5.0 breaking changes. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.0/050-breaking-changes.html

[84] Etherscan. (2021) Contract – 0x744d70FDBE2Ba4CF95131626614a1763DF805B9E. [Online]. Available: https://etherscan.io/address/0x744d70fdbe2ba4cf95131626614a1763df805b9e#code

[85] Status. (2018) Status - access a better web, anywhere. [Online]. Available: https://status.im/

[86] Etherscan. (2021) Contract – 0x58FA7c39893d97dF81A0BacC997958CE7199E22d. [Online]. Available: https://etherscan.io/address/0x58fa7c39893d97df81a0bacc997958ce7199e22d#code

[87] Ethereum. (2015) Eip-20: Erc-20 token standard. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

[88] Etherscan. (2021) Contract – 0xe367Fe584ADad878B9174ea4ddD8571878016811. [Online]. Available: https://etherscan.io/address/0xe367Fe584ADad878B9174ea4ddD8571878016811#code

[89] P. Gomes. (2019) ChainId vs NetworkId? how do they differ on Ethereum? [Online]. Available: https://medium.com/@pedrouid/chainid-vs-networkid-how-do-they-differ-on-ethereum-eec2ed41635b

[90] K. W. Jie. (2018) EIP712 is here: What to expect and how to use it. [Online]. Available: https://medium.com/metamask/eip712-is-coming-what-to-expect-and-how-to-use-it-bb92fd1a7a26

[91] Etherscan. (2021) Contract – 0xE0dCc70880bee5579DD93C317d272a4121A80117. [Online]. Available: https://etherscan.io/address/0xe0dcc70880bee5579dd93c317d272a4121a80117#code

[92] Etherscan. (2021) Contract – 0x84aC94F17622241f313511B629e5E98f489AD6E4. [Online]. Available: https://etherscan.io/address/0x84ac94f17622241f313511b629e5e98f489ad6e4#code

[93] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *In Proc. IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2017.

[94] costa group. (2021) A tool that applies super-optimization techniques to optimize ethereum's smart contracts. [Online]. Available: https://github.com/costa-group/syrup-python

[95] T. Chen, Y. Feng, Z. Li, H. Zhou, X. Luo, X. Li, X. Xiao, J. Chen, and X. Zhang, "GasChecker: Scalable analysis for discovering gas-inefficient smart contracts," *IEEE Transactions on Emerging Topics in Computing*, 2020.

[96] chriseth. (2016) Version 0.4.5. [Online]. Available: https://github.com/ethereum/solidity/releases/tag/v0.4.5

[97] Etherscan. (2021) Contract – 0xA43EBd8939D8328F5858119a3fb65f65c864c6Dd. [Online]. Available: https://etherscan.io/address/0xa43ebd8939d8328f5858119a3fb65f65c864c6dd#code

[98] Ethereum. (2021) Compiler input and output json description. [Online]. Available: https://solidity.readthedocs.io/en/latest/using-the-compiler.html#input-description

[99] S. Sarwar, R. Koretsky, and S. Sarwar, *Unix: The Textbook*. Pearson/Addison-Wesley, 2005. [Online]. Available: https://books.google.co.kr/books?id=IoQhAQAAIAAJ

[100] B. A. Kitchenham and S. L. Pfleeger, "Personal opinion surveys," in *Guide to advanced empirical software engineering*. Springer, 2008.

[101] Ethereum. (2021) Estimategas source code. [Online]. Available: https://github.com/ethereum/go-ethereum/blob/433f0919cc396cc565b38680abba9e4c6f4622f1/internal/ethapi/api.go#L986

[102] S. Overflow. (2021) Infinite gas warning in remix for a function. [Online]. Available: https://stackoverflow.com/questions/68625587/infinite-gas-warning-in-remix-for-a-function

[103] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *In Proc. International Conference on Software Engineering*, 2020.

[104] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *29th USENIX Security Symposium*, 2020.

[105] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "scompile: Critical path identification and analysis for smart contracts," in *International Conference on Formal Engineering Methods*. Springer, 2019, pp. 286–304.

[106] D. Wang, B. Jiang, and W. Chan, "Wana: Symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection," *arXiv preprint arXiv:2007.15510*, 2020.

[107] Y. Huang, B. Jiang, and W. Chan, "Eosfuzzer: Fuzzing eosio smart contracts for vulnerability detection," *arXiv preprint arXiv:2007.14903*, 2020.

[108] I. Ashraf, X. Ma, B. Jiang, and W. Chan, "Gasfuzzer: Fuzzing ethereum smart contract binaries to expose gas-oriented exception security vulnerabilities," *IEEE Access*, 2020.

[109] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," *arXiv preprint arXiv:1905.06944*, 2019.

[110] X. Wang, J. He, Z. Xie, G. Zhao, and S.-C. Cheung, "Contractguard: Defend ethereum smart contracts with embedded intrusion detection," *IEEE Transactions on Services Computing*, vol. 13, no. 2, pp. 314–328, 2019.

[111] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu, "Vultron: catching vulnerable smart contracts once and for all," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2019, pp. 1–4.

[112] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Transactions on Software Engineering*, 2020.

[113] E. Viglianisi, M. Ceccato, and P. Tonella, "A federated society of bots for smart contract testing," *Journal of Systems and Software*, p. 110647, 2020.

[114] J. Chen, "Finding ethereum smart contracts security issues by comparing history versions," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1382–1384.

[115] B. Gao, S. Shen, L. Shi, J. Li, J. Sun, and L. Bu, "Verification assisted gas reduction for smart contracts."

[116] M. M. A. Khan, H. M. A. Sarwar, and M. Awais, "Gas consumption analysis of ethereum blockchain transactions," *Concurrency and Computation: Practice and Experience*, p. e6679.

[117] A. A. Zarir, G. A. Oliva, Z. M. Jiang, and A. E. Hassan, "Developing cost-effective blockchain-powered applications: A case study of the gas usage of smart contract transactions in the ethereum blockchain platform," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–38, 2021.

[118] L. Marchesi, M. Marchesi, G. Destefanis, G. Barabino, and D. Tigano, "Design patterns for gas optimization in ethereum," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2020, pp. 9–15.

[119] T. Hukkinen, J. Mattila, K. Smolander, T. Seppala, and T. Goodden, "Skimping on gas–reducing ethereum transaction costs in a blockchain electricity market application," 2019.

[120] T. Chen, Z. Li, H. Zhou, J. Chen, X. Luo, X. Li, and X. Zhang, "Towards saving money in using smart contracts," in *In Proc. IEEE/ACM International Conference on Software Engineering*, 2018.

[121] X. Wang, H. Wu, W. Sun, and Y. Zhao, "Towards generating cost-effective test-suite for ethereum smart contract," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 549–553.

[122] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Don't run on fumes—parametric gas bounds for smart contracts," *Journal of Systems and Software*, vol. 176, p. 110923, 2021.

[123] E. Albert, J. Correas, P. Gordillo, G. Román-Díez, and A. Rubio, "Gasol: Gas analysis and optimization for ethereum smart contracts," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, Cham, 2020, pp. 118–125.

[124] D. Soto, A. Bergel, and A. Hevia, "Fuzzing to estimate gas costs of ethereum contracts," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 687–691.

[125] M. Marescotti, M. Blicha, A. E. Hyvärinen, S. Asadi, and N. Sharygina, "Computing exact worst-case gas consumption for smart contracts," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018, pp. 450–465.

[126] F. Ma, Y. Fu, M. Ren, W. Sun, Z. Liu, Y. Jiang, J. Sun, and J. Sun, "Gasfuzz: Generating high gas consumption inputs to avoid out-of-gas vulnerability," *arXiv preprint arXiv:1910.02945*, 2019.

[127] S. Bouraga, "An evaluation of gas consumption prediction on ethereum based on transaction history summarization," in *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS)*. IEEE, 2020, pp. 49–50.

[128] S. M. Werner, P. J. Pritz, and D. Perez, "Step on the gas? a better approach for recommending the ethereum gas price," in *Mathematical Research for Blockchain Economy*. Springer, 2020, pp. 161–177.

[129] G. A. Pierro, H. Rocha, R. Tonelli, and S. Ducasse, "Are the gas prices oracle reliable? a case study using the ethgasstation," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, 2020, pp. 1–8.

[130] D. Carl and C. Ewerhart, "Ethereum gas price statistics," *University of Zurich, Department of Economics, Working Paper*, no. 373, 2020.

[131] R. Mars, A. Abid, S. Cheikhrouhou, and S. Kallel, "A machine learning approach for gas price prediction in ethereum blockchain," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 156–165.

[132] F. Liu, X. Wang, Z. Li, J. Xu, and Y. Gao, "Effective gasprice prediction for carrying out economical ethereum transaction," in *2019 6th International Conference on Dependable Systems and Their Applications (DSA)*. IEEE, 2020, pp. 329–334.

[133] G. A. Pierro, H. Rocha, S. Ducasse, M. Marchesi, and R. Tonelli, "A user-oriented model for oracles' gas price prediction," *Future Generation Computer Systems*, vol. 128, pp. 142–157, 2022.

[134] R. Yang, T. Murray, P. Rimba, and U. Parampalli, "Empirically analyzing ethereum's gas mechanism," in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2019, pp. 310–319.

[135] A. Aldweesh, M. Alharby, and A. van Moorsel, "Performance benchmarking for ethereum opcodes," in *2018 IEEE/ACS 15th International Conference on Computer Systems and Applications (AICCSA)*. IEEE, 2018, pp. 1–2.

[136] 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 *International conference on information security practice and experience*. Springer, 2017, pp. 3–24.

[137] C. Liu, J. Gao, Y. Li, and Z. Chen, "Understanding out of gas exceptions on ethereum," in *International Conference on Blockchain and Trustworthy Systems*. Springer, 2019, pp. 505–519.

[138] C. Liu, J. Gao, Y. Li, H. Wang, and Z. Chen, "Studying gas exceptions in blockchain-based cloud applications," *Journal of Cloud Computing*, vol. 9, no. 1, pp. 1–25, 2020.

[139] S. Kim, J. Song, S. Woo, Y. Kim, and S. Park, "Gas consumption-aware dynamic load balancing in ethereum sharding environments," in *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W)*. IEEE, 2019, pp. 188–193.

[140] P. Zheng, Z. Zheng, X. Luo, X. Chen, and X. Liu, "A detailed and real-time performance monitoring framework for blockchain systems," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 2018, pp. 134–143.

[141] Z. Zheng, S. Xie, H.-N. Dai, W. Chen, X. Chen, J. Weng, and M. Imran, "An overview on smart contracts: Challenges, advances and platforms," *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020.

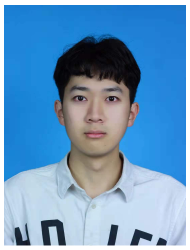
[142] S. N. Khan, F. Loukil, C. Ghedira-Guegan, E. Benkhelifa, and A. Bani-Hani, "Blockchain smart contracts: Applications, challenges, and future trends," *Peer-to-peer Networking and Applications*, pp. 1–25, 2021.

[143] X. Chen, P. Liao, Y. Zhang, Y. Huang, and Z. Zheng, "Understanding code reuse in smart contracts," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 470–479.

[144] M. Zhang, P. Zhang, X. Luo, and F. Xiao, "Source code obfuscation for smart contracts," in *2020 27th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2020, pp. 513–514.

Zhou Liao received his B.S. degree from University of Electronic Science and Technology of China (UESTC). Now he is a master student in UESTC. School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chendgu, Sichuan, China, (e-mail: liaozhou98@qq.com).

Shuwei Song received his B.S. degree from UESTC. Now he is a PhD student in UESTC. School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chendgu, Sichuan, China, (e-mail: shuwei@std.uestc.edu.cn).



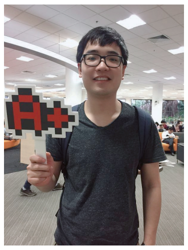
Hang Zhu received his B.S. degree from UESTC. Now he is a master student in UESTC. School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chendgu, Sichuan, China, (e-mail: 568991738@qq.com).

Xiapu Luo is an associate professor in the Department of Computing, The Hong Kong Polytechnic University. His current research interests include Mobile/IoT Security and Privacy, Blockchain and Smart Contracts, Network Security and Privacy, and Software Engineering. His work appeared in top venues in the areas of security, software engineering and networking. He has received eight best paper awards (e.g., INFOCOM'18, ISSRE'16, etc.) and an ACM SIGSOFT Distinguished Paper Award from ICSE'21.

Zheyuan He received his B.S. degree from BESTI. Now he is a master student in UESTC. School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chendgu, Sichuan, China, (e-mail: ecjgvmhc@gmail.com).

Renkai Jiang received his B.S. degree from UESTC. Now he is a master student in UESTC. School of Computer Science and Engineering, University of Electronic Science and Technology of China, Chendgu, Sichuan, China, (e-mail: Renkai_J@std.uestc.edu.cn).

Ting Chen received his PhD degree from UESTC, China, 2013. He is a Professor in the School of Computer Science and Engineering in UESTC. His research interest focuses on blockchain, smart contract and software security. He has published tens of high quality papers in prestigious conferences and journals. His work received several best paper awards, including INFOCOM 2018 best paper award.



Jiachi Chen received the B.S. degree in Institute of Service Engineering, HangZhou Normal University, China in 2016. Currently, he is a PhD student in Monash University. He got Sliver Award in 12th Zhejiang Undergraduate ACM Program Design Competition. His research interest includes data mining and program analysis.

Tao Zhang received the BS degree in automation, the MEng degree in software engineering from Northeastern University, China, and the PhD degree in computer science from the University of Seoul, South Korea. After that, he spent one year with the Hong Kong Polytechnic University as a postdoctoral research fellow. Currently, he is an associate professor with the School of Computer Science and Engineering, Macau University of Science and Technology (MUST). Before joining MUST, he was the faculty member of Harbin Engineering University and Nanjing University of Posts and Telecommunications, China. He published more than 60 high-quality papers at renowned software engineering and security journals and conferences such as TSE, TIFS, TDSC, TR, ICSE, etc. His current research interests include AI for software engineering and mobile software security. He is a senior member of IEEE and ACM.

Xiaosong Zhang received his PhD degree from UESTC, 2011, and is now a Professor in the School of Computer Science and Engineering in UESTC. His research focuses on network security, AI security and blockchain security. He is the director of Cybersecurity Institute of UESTC. He published more than 100 papers about cybersecurity, blockchain and AI secuirty. He received the First prize of national science and technology progress award 2019, and the second prize of national science and technology progress award 2021.