# Automatic Identification of Crash-inducing Smart Contracts

Chao Ni*
School of Software Technology
Zhejiang University, China
chaoni@zju.edu.cn

Cong Tian
College of Computer Science and Technology
Zhejiang University, China
tianc43@zju.edu.cn

Kaiwen Yang
College of Computer Science and Technology
Zhejiang University, China
kwyang@zju.edu.cn

David Lo
Singapore Management University
Singapore
davidlo@smu.edu.sg

Jiachi Chen
School of Software Engineering
Sun Yat-sen University, China
chenjch86@mail.sysu.edu.cn

Xiaohu Yang
College of Computer Science and Technology
Zhejiang University, China
yangxh@zju.edu.cn

***Abstract*—Smart contract, a special software code running on and resided in the blockchain, enlarges the general application of blockchain and exchanges assets without dependence of external parties. With blockchain's characteristic of immutability, they cannot be modified once deployed. Thus, the contract and the records are persisted on the blockchain forever, including failed transactions that are caused by runtime errors and result in the waste of computation, storage, and fees. In this paper, we refer to smart contracts which will cause runtime errors as crash-inducing smart contracts. However, automatic identification of crash-inducing smart contracts is limited investigated in the literature. The existing approaches to identify crash-inducing smart contracts are either limited in finding vulnerability (e.g., pattern-based static analysis) or very expensive (e.g., program analysis), which is insufficient for Ethereum.**

**To reduce runtime errors on Ethereum, we propose an efficient, generalizable, and machine learning-based crash-inducing smart contract detector, CRASHSCDET, to automatically identify crash-inducing smart contracts. To investigate the effectiveness of CRASHSCDET, we firstly propose 34 static source code metrics from four dimensions (i.e., *complexity metrics*, *count metrics*, *object-oriented metrics*, and *Solidity-specific metrics*) to characterize smart contracts. Then, we collect a large-scale dataset of verified smart contracts (i.e., 54,739) and label these smart contracts based on their execution traces on Etherscan. We make a comprehensive comparison with three state-of-the-art approaches and the results show that CRASHSCDET can achieve good performance (i.e., 0.937 of F1-measure and 0.980 of AUC on average) and statistically significantly improve the baselines by 0.5%-60.4% in terms of F1-measure and by 41.2%-44.3% in terms of AUC, which indicates the effectiveness of static source code metrics in identifying crash-inducing smart contracts. We further investigate the importance of different types of metrics and find that metrics in different dimensions have varying abilities to depict the characteristic of smart contracts. Especially, metrics belonging to the "*Count*" dimension are the most discriminative ones but combining all metrics can achieve better prediction performance.**

***Index Terms*—Crash-inducing Smart Contract, Static Source Code Metric, Quality Assurance, Ethereum, Machine Learning**

*Chao Ni is the corresponding author.

## I. INTRODUCTION

A smart contract is a program running on the blockchain [1], and its correct execution is enforced by the consensus protocol. The smart contract is identified by an address (a 160-bit identifier) and its code is persisted on the blockchain. The most attractive property of the smart contract is that it can eliminate the need of a trusted third party in multi-party interactions, enabling all parties to participate in secure peer-to-peer transactions.

Although smart contracts have been widely deployed on the Ethereum platform [2], not all contract transactions are executed successfully. Many transactions are reverted by the EVM due to runtime errors [3]. These failed transactions will not change global state and are a waste of computation. In addition, considering the immutability of the blockchain, all processed transactions including failed ones, are stored permanently in the blocks and waste storage. That is, failed transactions caused by runtime errors on Ethereum may have some side-effects: ❶ **causing the consumption of storage and the waste of fees**; ❷ **affecting the execution efficiency of Ethereum** [3]. In this paper, we refer to smart contracts which will cause runtime errors as crash-inducing smart contracts. Identifying issues in smart contracts in advance can avoid failure-inducing transactions submitted to Ethereum and can help improve the efficiency of Ethereum.

There are a few feasible methods to test whether a smart contract is crash-inducing or not, such as fuzzing technology [4, 5] and tools based on static analysis [6–8]. However, these existing methods have some limitations. ① As for fuzzing technology, it generates a large number of test cases to detect crash-inducing smart contracts. However, generating enough test cases to identify crash-inducing smart contracts is time-consuming and high-quality predefined rules for generating testing cases are also needed. As for tools based on static analysis [6–8], there exist two major kinds: static analysis on patterns and static analysis on a specific version of Ethereum. ② For the former one, tools can only identify those crash-inducing smart contracts with existing patterns. For uncovered patterns, these tools can do nothing. ③ For the latter one, these static analysis tools are based on a specific version of Ethereum. However, Ethereum often adds or changes some instructions [9] through a hard fork, which will make the existing tools invalid. The existing approaches to identify crash-inducing smart contracts are either limited in finding vulnerability (e.g., pattern-based static analysis) or very expensive (e.g., program analysis) [10], which is insufficient for Ethereum. Besides, detecting crash-inducing smart contracts is not trivial, and an effective and generalizable method to identify crash-inducing smart contracts is urgently needed. Identifying crash-inducing smart contracts in advance can help to avoid the loss of gas, decrease the number of invalid transactions, and improve the efficiency of Ethereum.

In the scenario of software quality assurance (SQA) [11–14], static source code metrics (e.g., Halstead metrics [15], McCabe metrics [16]) are treated as important roles and many researchers have investigated the relationship between static source code metrics and the software quality (e.g., software crash). Many approaches are proposed based on the static source code metrics and prior work confirms these metrics are useful for quality assurance [17]. Thus, we think the smart contracts can also benefit from static source code metrics and we can identify crash-inducing smart contracts in advance with static source code metrics.

In this paper, to reduce runtime errors on Ethereum, we propose an efficient, generalizable, and machine learning based crash-inducing smart contract detector, CRASHSCDET, to automatically identify crash-inducing smart contracts. To investigate CRASHSCDET's effectiveness, we firstly, inspired by previous work [14–16] in software quality assurance, propose 34 static source code metrics from four dimensions (i.e., *complexity metrics*, *count metrics*, *object-oriented metrics*, and *Solidity-specific metrics*) to measure a smart contract. We adopt widely used metrics in SQA, which are general ones. Besides, we also propose some specific metrics for smart contracts since we think the characteristic (e.g., gas-driven) of the smart contract is also a good indicator of quality. Then, we totally collect 54,739 smart contracts on Etherscan and label them according to their execution traces of transaction. The results show that CRASHSCDET can achieve good performance (i.e., 0.937 of F1-measure and 0.980 of AUC on average) and statistically significantly improve the baselines by 0.5%-60.4% in terms of F1-measure and by 41.2%-44.3% in terms of AUC, which indicates the effectiveness of static source code metrics in identifying crash-inducing smart contracts. Finally, we make an analysis of the studied smart contracts and further analyze the importance of metrics from different dimensions. We find that metrics in different dimensions have varying abilities to depict the characteristic of smart contracts and "*count metric*" is the most important one. Eventually, our work makes the following contribution.

- ■ We propose the problem of automatic identification of crash-inducing smart contracts, and propose 34 static source code metrics from four dimensions to characterize a smart contract. To the best of our knowledge, this paper is the first to detect crash-inducing smart contracts.
- ■ We collect a large-scale dataset (i.e., 54,739 verified contracts with source codes) and label these contracts by crawling the transaction running traces of smart contracts since there is no dataset available today.
- ■ We confirm the usefulness of static source code metrics in identifying crash-inducing smart contracts and also propose an efficient and generalizable crash-inducing smart contract detector named CRASHSCDET. The replication package of dataset and code is publicly available [18].
- ■ We investigate the importance of different types of metrics, and we find that metrics in different dimensions have varying abilities to depict the characteristic of smart contracts. We find that the metrics in the "Count" dimension are the most discriminative dimension but combining all the metrics from all dimensions can achieve better performance.

**Paper Organization**. Section II introduces the background of Ethereum, EVM, smart contract and error types in EVM. Section III illustrates the framework of our approach. Section IV describes the experimental settings, including the dataset, the definition of proposed metrics, the baselines, the performance measures and the validation setting. Section V presents the research question and analyzes the experimental results. Section VI lists the potential threats to validity. Section VII briefly reviews the related work and Section VIII concludes this paper.

## II. BACKGROUND

### A. Ethereum

Bitcoin is the first blockchain-based cryptocurrency system and was introduced in 2008, which brings enormous potential of blockchain to the world. The major limitation of Bitcoin is that it only allows users to encode non-Turing-complete scripts to execute transactions and consequently reduces its ability. Then, Ethereum was invented by Vitalik et al. in 2015 to address this limitation. Vitality et al. also brought a new technology named smart contracts to the world. Nowadays, Ethereum has become the second most popular blockchain system, and meanwhile, it is the most popular platform on which smart contracts can run [19].

### B. Ethereum Virtual Machine (EVM)

EVM is a quasi-Turing complete machine that executes smart contracts and performs error handling to avoid machine failures. It is implemented as a virtual stack machine with a pre-defined instruction set (i.e., opcode [20]). When a user transaction triggers a smart contract, the EVM uses the input provided by the user and the smart contract state to execute the contract using gas as fuel. Gas is used as the upper limit of the EVM execution workload to prevent the endless execution of the contract.

### C. Smart Contract

Ethereum smart contracts can be written in high-level programming languages and be compiled into EVM bytecode.

Solidity is the most popular one and can be used to develop smart contracts on Ethereum. When a smart contract is created, its bytecode is stored in the contract account to facilitate transactions. Smart contracts cannot be modified once deployed since the immutability of blockchain, but they can be deleted using a *self-destruct* opcode. To identify a contract address, a unique 40 bytes hexadecimal hash value is needed. Since Ethereum is a permission-less network, everyone can send a transaction with both the contract name and corresponding input to a specific contract address to call the contract.

### D. *Error Type*

The EVM has defined six runtime error types [21]:

- **Out of Gas**. Out of gas error is caused by insufficient gas when you don't provide enough gas to execute a transaction, or provided gas is not enough to complete a transaction.
- **Revert**. Transaction revert occurs when a smart contract wants to do something that is undefined or unauthorized in the logic of smart contract in a transaction. Thus, such a transaction will be reverted and EVM returns an error.
- **Invalid Opcode**. Invalid opcode error occurs when a smart contract tries to call a code that doesn't exist.
- **Invalid Jump**. Invalid jump occurs when a smart contract tries to call a nonexistent function or when a smart contract uses assembly language and points to wrong memory.
- **Stack Overflow**. Stack overflow error occurs when a smart contract calls a function recursively with no stop condition. In Solidity, the stack has at most 1,024 frames, which means a function can call itself at most 1,024 times. Therefore, stack overflow will occur if a function is called more than 1,024 times.
- **Stack Underflow**. Stack underflow error occurs in assembly language if a smart contract tries to pop a variable that does not exist.

Notice that, after analyzing the transaction log information on Ethereum, we find that both stack overflow and stack underflow are grouped into the type of "Out of Stack". Invalid opcode also has the name of "Bad Instruction", and "Invalid Jump" also the name of "Bad Jump Destination". We use these names interchangeably in this paper.

## III. Approach

To automatically identify crash-inducing smart contracts, we propose CrashSCDet with two phases: ① detector building phase and ② detector application phase. The framework of CrashSCDet is illustrated in Fig. 1.

### A. *Detector building phase*

In the detector building phase, CrashSCDet firstly collects all transactions of each verified smart contract deployed on Ethereum and labels the smart contracts with their status of transactions. The details can be found in Section IV-B. Then, we design a set of metrics to characterize the smart contract. In particular, we totally design 34 static source code metrics which can be grouped into four categories: *Complexity Metric*, *Count Metric*, *Object Oriented Metric* and *Solidity-specific Metric*. Details can be referred to in Section IV-A. After that, CrashSCDet converts source codes of smart contracts into tabular data, and then builds a detector after pre-processing (e.g., re-sampling) on training data. By default, we adopt Random Forest (RF) as the basic classifier with pre-defined parameters and we use the implementation in scikit-learn [22]. Random Forest [23], an ensemble approach, is specifically designed for decision tree classifier. It fits many random decision trees (e.g., C4.5) on various sub-samples of the dataset and combines multiple decision trees for classification. Random Forest has the following advantages: 1) it usually has high precision, and the importance of metrics can be generated automatically; 2) it can mitigate the over-fitting problem and is insensitive to outliers since it combines many simple decision trees that are learned differently.

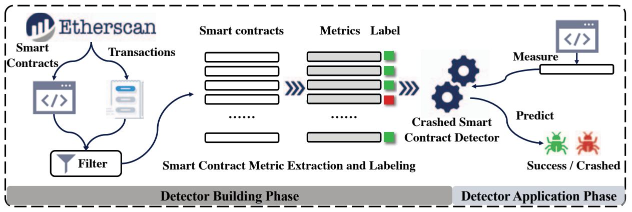


**Fig. 1:** The framework of CrashSCDet

### B. *Detector application phase*

In the defector application phase, CrashSCDet firstly converts the source code of a smart contract into pre-defined metrics. Then, the converted data (i.e., vector-like data) is fed into CrashSCDet, which will output the corresponding status of the smart contract. That is, whether the smart contract will crash during the process of a transaction.

## IV. Study Setup

This section firstly introduces the definition of metrics for characterizing smart contracts. Then, it presents how to collect and label smart contracts. Following that, the experimental setting is presented, including the introduction of baselines, the validation settings, the performance measures and the statistical analysis.

### A. *Defining Metrics For Smart Contract*

As static source code metrics (e.g., Halstead metrics [15], McCabe metrics [16], Chidamber & Kemerer [24]) for other object-oriented programming languages play a very important role in various SQA activities (e.g., software quality, maintainability) and prior work [14, 25, 26] have indicated the advantages of these metrics, we expect that the same would be true for smart contracts. Meanwhile, Solidity widely used to develop smart contracts is also an object-oriented high-level programming language. We consider all these important metrics used in previous work to represent a smart contract.

Inspired by existing work [14–16], we totally define 34 static source code metrics, of which 22 metrics are programming language-independent ones and 12 metrics are Solidify-specific ones. Following existing work [27], we further divide the programming-independent metrics into three dimensions: *complexity metrics*, *count metrics* and *object-oriented metrics*.

To represent a contract with several metrics, we implement our prototype tool on top of *py-solc-x* [28] and *py-solc-ast* [29]. *py-solc-x* is a python wrapper and version management tool for the *solc* (i.e., the Solidity compiler), while *py-sol-ast* is a tool for exploring the $solc$ abstract syntax tree (AST). We calculate the metrics of contract either on the Solidity source codes directly or by implementing visitors to collect the necessary information from the built AST. We introduce these metrics as follows.

❒ **Complexity Metrics** are used to measure the number of paths through a program, which indicates the code complexity [16]. Intuitively, the more complex a piece of code is, the higher probability of it being a defective code. Complexity dimension has six metrics: *AvgCyclomatic*, *MaxCyclomatic*, *SumCyclomatic*, *MaxInheritanceTree*, *MaxNesting* and *CountContractCoupled*. The introductions to them are as below.

**AvgCyclomatic** & **MaxCyclomatic** & **SumCyclomatic**. McCabe's cyclomatic complexity [16] represents the sum of the number of branching statements and 1. A function in a smart contract is represented as a function node in AST. We calculate the sum of the number of branching statements from two aspects: 1) traversing and counting all the branch statements (e.g., *if*, *while*, *do while*, and *for*) in the subtree of AST under the appropriate function node; 2) calculating the total number of "&&", "||", and "? : " in source code file. Suppose there are $n$ functions in a Solidity file, we have $n$ numbers of McCabe's cyclomatic complexity for all functions (i.e., $M_1, M_2, ..., M_n$). Thus, *MaxCyclomatic* value for a Solidity file is the max value of McCabe's cyclomatic complexity among all functions in a Solidity file (i.e., max($M_1, M_2, ..., M_n$)). *SumCyclomatic* value for a Solidity file is the sum value of McCabe's cyclomatic complexity among all functions in a Solidity file (i.e., sum($M_1$, $M_2$, ..., $M_n$)). *AvgCyclomatic* value for a Solidity file is the average value of McCabe's cyclomatic complexity among all functions in a Solidity file (i.e., sum($M_1$, $M_2$, ..., $M_n$)/n).

**MaxInheritanceTree**. The maximum depth of a contract in the inheritance tree metric represents the maximum depth of inheritance, which indicates how deep the main contract is in the inheritance tree. In our studied contracts, there may have a few contracts in a Solidity file, but only one contract is the main contract and other contracts are the helper contracts or inherited contracts. A recursive algorithm is used to calculate this metric. We set a *MaxInheritanceTree* value to the main contract that is the maximal *MaxInheritanceTree* measure of its parents plus 1. Notice that we always take the deepest path through the inheritance tree since the Solidity programming language supports multiple inheritances. The *MaxInheritanceTree* values of parent nodes are calculated recursively until a node has no parents, and the corresponding *MaxInheritanceTree* value is 0.

**MaxNesting**. The nesting level metric indicates the deepest nesting level of the control structures within the functions in a Solidity file. To calculate the nesting levels of a single function, we access all the statements in the subtree rooted by the function and count the number of branching statements on the path from the statement to the function definition traversing the parent nodes. Therefore, the *MaxNesting* value of a Solidity file is the maximum value among all nesting level values calculated for the statements in a function.

**CountContractCoupled**. The coupling between contracts metric measures the count of the contracts coupled with a certain contract. When a function of one contract calls a function of another contract or accesses a variable of another contract, we consider the two contracts to be coupled. This metric is used to reflect how contracts are connected to each other and it reflects how many other types of contracts are used (i.e., as state variable type, local variable type, function parameters, etc.) by a particular contract. We calculate *CountContractCoupled* of a contract *C1* as the size of the collection of contracts that reference *C1* or are referenced by *C1*.

❒ **Count Metrics** are used to represent code characteristic from the physical size (e.g., functional statements, blank lines, or code comments), which are good indicators of faults as confirmed by Gyimothy et al. [30]. Intuitively, the larger code-size of a code, the more probability of a defective code. The count metric dimension is also composed of six metrics: *CountLineCode*, *CountLineCodeExe*, *CountLineComment*, *CountStmt*, *CountLineBlank* and *RatioCommentToCode*. The introduction to these metrics are as follows:

**CountLineCode**. The source lines of code metric denotes the number of source code lines in a Solidity file. It is calculated based on the starting and ending line number of the file after pre-processing.

**CountLineCodeExe**. The number of executable lines of code metric counts the non-empty and non-comment lines in a Solidity source code file. That is, those lines, containing actual statements, are only counted. To calculate this metric, we scan the source code line by line and filter out all the empty and comment lines to get the *CountLineCodeExe* metric. In Solidity programming language, there are two types of methods to comment on the source code: (1) single-line comments ("//[/]") and (2) multi-line comments ("/*...*/"). Therefore, we consider all lines to be comment lines if they start with "/*", "*", "//[/]" or end with "*/".

**CountLineComment**. The lines with comments metric is the number of comment lines in a Solidity source code file. It is calculated from the Solidity source file with the heuristic described above at *CountLineCodeExe* metric.

**CountStmt**. The number of statements metric is a simple size metric and it counts how many statements there are in a Solidity source code file. We calculate this metric by counting the number of statement definition in parsed AST of a specific Solidity source code file.

**CountLineBlank**. The blank lines of code metric is the number of blank lines in a Solidity source code file. To calculate this metric, we scan the source code line by line and search all the empty lines to get the *CountLineBlank* metric.

**RatioCommentToCode**. The ratio of comment lines to code lines metric in a solidity source code file is defined as the ratio of *CountLineComment* and *CountLineCode*.

❒ **Object-oriented Metrics** are used to characterize the quality of source code from a high-level view. The object-oriented approach centers around modeling the real world in terms of its objects, which is in contrast to the traditional approaches that emphasize a function-oriented view that separates data and procedures [24]. Prior work [31] has confirmed the relationship between object-oriented metrics and the code quality. The object-oriented metric dimension includes ten metrics: *CountContractBase*, *CountDependence*, *CountContract*, *CountTotalFunction*, *CountFunctionExternal*, *CountFunctionPublic*,*CountFunctionInternal*, *CountFunctionPrivate*, *CountPublicVariable* and *CountVariable*. The introduction to these metrics are as follows:

**CountContractBase**. The base contract metric indicates the number of direct base contracts of the main contract. We calculate this metric from the parsed AST of a specific Solidity source code file by counting the directly inherited basic contracts with the help of *py-solc-ast*.

**CountDependence**. The number of dependence metric indicates the number of direct dependence or indirect dependence of the main contract in a Solidity source code file. In Solidity programming language, a contract can inherit from multi contracts, and the inherited contracts may also inherit from other contracts. We calculate all the number of immediate base contracts and the indirect inherited contracts.

**CountContract**. This metric measures the number of contracts in a Solidity source code file. In a Solidity source code file, there may have a few contract definitions, in which, only one contract is treated as the main contract, the others are treated as non-main contracts.

**CountFunctionInternal** & **CountFunctionExternal** & **CountFunctionPrivate** & **CountFunctionPublic** & **CountTotalFunction**. In Solidity, there are four types of visibility for functions and state variables: *external*, *public*, *internal*, or *private*. Functions defined in Solidity have to be specified as one of the four types. However, for state variables, *external* is not possible. The external function is part of the contract interface, which means they can only be called from other contracts and via transactions. The public function is also part of the contract interface and can be either called internally or via messages. For public state variables, an automatic getter function is generated in a contract. For internal function and state variables, they can only be accessed internally (i.e., within the current contract or contracts deriving from it). For private function and state variables, they are only visible for the contract they are defined in and not in derived contracts. To calculate these metrics in a Solidity source code file, we sum up the corresponding number of definitions of functions which are *external*, *public*, *internal*, or *private* respectively based on function nodes in the AST. We also sum up the total number of functions in a Solidity source code file as the value of *CountTotalFunction*.

**CountPublicVariable** & **CountVariable**. We calculate the number of variables in a Solidity source code file as the value of *CountVariable* and calculate the number of public variables in a Solidity file as the value of *CountPublicVariable*.

❒ **Solidity-specific Metrics**. Different programming languages have different characteristics and consequently have different impacts on the quality of code [32–34]. These aforementioned metrics are general metrics. However, Solidity is a contract-oriented, high-level language for implementing smart contracts. It is designed to target the EVM, which is quite sensitive to the resource (e.g., gas). Almost all of the operations (e.g., transferring, storing) in a smart contract are driven by sufficient gas. The gas refers to the unit that measures the amount of computational effort required to execute specific operations on the Ethereum network. The gas is valuable and can be converted into real-world currency. Therefore, considering the characteristic of smart contract, we propose some heuristic metrics which are specific to Solidify programming language.

Firstly, since the gas-driven characteristic of Ethereum, all instructions in the smart contract need to consume gas, for example, transferring operation, storing corresponding data, calling other smart contracts, recording the running logs, and so on. Different instructions cost varying numbers of gas. Therefore, the more number of instructions in a smart contract, the more gas will be consumed. This increases the probability of a smart contract crash due to the exhaustion of gas. Thus, we define the eight metrics (e.g., *NOSV*, *NOMap*, *NOPay*, *NOE*, *NOMod*, *NOT*, *NOC* and *NODC*) to identify this characteristic.

Secondly, since the immutability characteristic of Ethereum, we cannot modify any code in a smart contract deployed on Ethereum even it contains bugs or servers as a library for other smart contracts. Besides, for a specific smart contract, we cannot make any modifications to what is implicitly or explicitly claimed to be unchangeable. As for this characteristic, it will increase the risky of crash. Therefore, we define two metrics (e.g., *NOSF* and *NOL*) to identify this characteristic.

Finally, the default action in a smart contract will have a large influence on its behavior. For example, the default "fallback" function can only accept calls with data but reject calls with gas. Therefore, the smart contract must define its own "fallback" function and mark it with "payable" feature. We define "*SDFB*" to identify this characteristic. At the same time, we think a good design of code struct (e.g., modularization) will impact the readability of code and consequently impact the quality of code, therefore we simply define "*NOI*" metric to identify its good structure.

In summary, we propose 12 Solidity-related metrics and introduce them as follows.

**NOSV** is short for the number of storage variables. The EVM has three areas where it can store data: *storage*, *memory* and the *stack*. Each account in Ethereum has a data area called *storage*, which is persistent between function calls and transactions. However, it is comparatively costly to read, and even more to initialize and change storage. The number of storage variable metric measures how many variables stored in the *storage* area are defined in a Solidity source code file. Besides, there are three data types (i.e., *mapping*, *array*, *struct*) are *storage* variable by default. Different from other programming languages, all the storage variables in Solidity will cause the consumption of gas. Therefore, the more storage variable defined the higher is the probability of a smart contract to crash. Therefore, we calculate the number of these typed variables in a Solidity source code file.

**NOMap** is short for number of mapping variables. Mapping type is one of the most used data types in Solidity programming language, which is similar to hashtable. However, there is no inner state variable to record the number of elements in it and the Mapping type is not supported to visit all elements like a "foreach loop" operation. Therefore, we think this type of variable will increase the risk of a smart contract to crash due to its peculiar usage and it will also cost more gas consumption for recording the length and key of the Mapping. The number of mapping variable metric measures how many variables are defined as the type of mapping in a Solidity source code file.

**NOPay** is short for number of payable functions. Payable functions provide a mechanism to receive funds in ethers to a contract and they are annotated with the *payable* keyword. However, a pay operation successfully executes with at least two basic conditions: sufficient gas from the sender and valid receiver. Therefore, the smart contract will crash if the two conditions are not satisfied. The number of payable function metric measures how many payable functions are defined in a Solidity source code file.

**NOE** is short for number of events. Events in Solidity is the wrapper of EVM's logging functionality. Applications can subscribe and listen to these events through the RPC interface of an Ethereum client to get the corresponding messages. However, the event also needs gas to drive it. The more usage of events, the more the consumption of gas is. Therefore, the number of events metric measures how many events are defined in a Solidity source code file.

**NOMod** is short for number of modifiers. Modifiers, a wrapper of function, can change the behavior of functions in a declarative way. For example, we can use a modifier to automatically check an expected condition prior to executing a specific function. The number of modifiers measures how many modifiers are defined in a Solidity source code file and it also consumes gas.

**NOT** is short for the number of transfer operations. The transfer function is an attribute of contract address, which can transfer the ether from one contract to another one. The transfer function consumes gas in two ways: one is the amount of gas the sender wants to send to the receiver, and another is the amount it needs to complete the operation. The **NOT** metric measures how many times the transfer function is invoked in a Solidity source code file.

**NOC** represents the number of call operations. Contracts can call other contracts through message calls. Message call is a low-level call method working on raw addresses in EVM and it can be used as a function call, ether sending, and so on. Every time a Solidity contract calls a function of another contract, it does so by producing a message call, which will cost a large number of units of gas. The number of message calls metric measures how many message calls are used in a Solidity source code file.

**NODC** is short for number of delegatecall operations. There exists a special variant of a message call, named delegatecall which is the same as a message call apart from the fact that the code at the target address is executed in the context of the calling contract and message's sender and message's value do not change their values. Similar to message call, it will also cost a large number of units of gas. The number of delegatecalls metric measures how many delegatecalls are used in a Solidity source code file.

**NOSF** is short for number of static functions. Static functions are expected not to modify the state of Ethereum. Two types of functions are declared not to modify the state: *pure* and *view*. Any instructions in static function try to change the state of Ethereum will cause a smart contract crash. Therefore, we calculate the number of static function metric by counting the number of *pure* or *view* functions in the source code file.

**SDFB** represents whether a smart contract uses self-define Fallback function. A contract can have one fallback function at most. The fallback function is executed on a call to the contract if none of the other functions match the given function signature, or if no data is supplied at all and there is no receive Ether function. The fallback function always receives data but rejects gas by default. To receive the gas, it must be marked as "payable". The self-define fallback function metric measures whether a contract defines a fallback function by itself.

**NOL** is short for number of libraries. Libraries are also similar to contracts, but their purpose is that they are deployed only once at a specific address and their code can be reused. Functions in a library can only be called directly if they do not modify the state since libraries are assumed to be stateless. However, the quality of the library is not under control and the code of the library cannot be modified, the smart contracts will have potential risk in the quality if there exists risky code in those used libraries. The number of library metric measures how many libraries are used in a Solidity source code file.

**NOI** is short for number of interfaces. Interfaces are similar to abstract contracts, which decouple the definition of a contract from its implementation, providing better extensibility and self-documentation and facilitating patterns like the template method and removing code duplication. However, interfaces cannot have any functions implemented. We believe a good design of code structure will improve the code readability and consequently improve the code quality. We simply think the usage of the interface will modularize the smart contract and consequently improve the quality of smart contract. The number of interface metric measures how many interfaces are defined in a Solidity source code file.

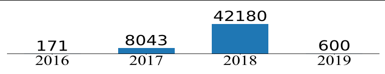
### *B. Collecting and Labeling Smart Contract*

**Smart contract collection.** We totally collect 54,739 smart contracts with source code written in Solidity programming language [35] from *Etherscan* [36]. The source codes of these contracts are submitted to *Etherscan* by their owners and are verified by *Etherscan*. We do not search on GitHub for smart contracts since we only want to collect those contracts deployed on the Ethereum network. The deployed contracts have meaningful transactions and some of those deployed contracts have verified source codes, which indicates that the functionality of the Solidity source code and the deployed EVM bytecode is manually compared and validated. We crawl and download these validated source codes of deployed smart contracts monitored by the Etherscan. Thus, we can make sure that the source codes we analyzed are actually the same as the contract being deployed on the Ethereum network. For further research on crash-inducing smart contracts, we set up the following exclusion criteria:

- **Remove contracts that have no transactions**. No transactions, no information for analyzing the execution quality of contracts (622 contracts).
- **Remove self-destruct contracts**. Self-destructed contracts can no longer run on Ethereum, so they cannot provide useful running information for future research.
- **Compilation error**. We cannot extract AST-based code metrics for smart contracts with compliation errors (e.g., “0xe9e2e188cdc5fcbc8fd2cdf553a98b43514241b4”).
- **Low Solidity version**. Contracts that can only be complied with “$<= 0.4.11$” Solidity version should be removed since they are not compatible with the new compiler and will gradually be abandoned.

After filtering by these criteria, 3,745 contracts are discarded and we finally obtain 50,994 contracts for this study.

**TABLE I:** Statistics of the analyzed smart contracts.

| Sol Files | Contract | Library | Interface | Event | Modifier | LOC |
|---|---|---|---|---|---|---|
| Total | 225,918 | 32,165 | 10,927 | 219,657 | 99,395 | 17,776,799 |
| Avg./sol File | 4.43 | 0.63 | 0.21 | 4.31 | 1.95 | 348.61 |
| Submitted for verification at Etherscan.io |  | | | | | |

**Smart contract labeling.** Labeling contracts as crash-inducing ones or not is extremely important. As Ethereum records all transactions and execution status of smart contracts, we use API provided by $Etherscan$ for extracting the execution traces of each transaction. A contract may have hundreds or thousands of transactions and each transaction can be treated as the test-case for a contract. A contract is labeled as a crash-inducing one if there exists one transaction of all transactions related to such a contract being executed incorrectly. A contract is labeled as a non-crash-inducing one if and only if all transactions related to such a contract are executed correctly.

The statistical information of the dataset is shown in Table I, including the statistics of smart contracts, libraries, interfaces, events, modifiers and LOCs. We totally analyze 225,918 sub-contracts (including main contract and its depended contracts), 32,165 libraries, 10,927 interfaces, 219,657 events, 99,395 modifiers and 17,776,799 LOCs. The second row shows the average statistical information of each Solidity file.

Table II shows the statistics from the side of the manually defined metrics for all the studied smart contracts. We can also draw a few general conclusions including but not limited to the following ones: (1) the complexity of smart contracts is low since *AvgCyclomatic* equals to 1.34, *MaxNesting* equals to 1.45 as well as *CountContractCoupled* equals to 0.53. (2) smart contracts are relatively easy to read given their size (i.e., *CountLineCodeExe* is 198) and detailed comments (i.e., *RatioCommentToCode* is 0.24). (3) smart contracts will cause more storage space in Ethereum (*NOSV* is 16.31). (4) half of smart contracts may support the transferring of Ether since average and median value of *NOPay* is 1.1 and 1, respectively.

**TABLE II:** Statistics of the calculated metric values.

| Dim. | Metric | Mean | Std. | Min | Q1 | Q2 | Q3 | Max |
|---|---|---|---|---|---|---|---|---|
| Complexity | AvgCyclomatic | 1.34 | 0.66 | 0 | 1.05 | 1.15 | 1.47 | 63.20 |
| | MaxCyclomatic | 3.64 | 5.96 | 0 | 2 | 2 | 4 | 611 |
| | MaxInheritanceTree | 1.47 | 1.32 | 0 | 0 | 1 | 2 | 10 |
| | MaxNesting | 1.45 | 1.82 | 0 | 1 | 1 | 2 | 125 |
| | SumCyclomatic | 29.72 | 33.53 | 0 | 12 | 18 | 33 | 632 |
| | CountContractCoupled | 0.53 | 0.88 | 0 | 0 | 0 | 1 | 17 |
| Count | CountLineCode | 348.61 | 434.73 | 2 | 132 | 223 | 394 | 11,706 |
| | CountLineCodeExe | 198.07 | 240.90 | 2 | 77 | 118 | 224 | 5,196 |
| | CountLineComment | 97.11 | 183.53 | 0 | 17 | 56 | 112 | 9,347 |
| | CountStmt | 78.06 | 90.96 | 0 | 29 | 47 | 88 | 1,424 |
| | CountLineBlank | 59.09 | 76.56 | 0 | 20 | 37 | 67 | 3,711 |
| | RatioCommentToCode | 0.24 | 0.15 | 0 | 0.12 | 0.26 | 0.35 | 0.95 |
| Object-Oriented | CountContractBase | 4.43 | 3.79 | 0 | 2 | 3 | 5 | 80 |
| | CountDependence | 3.01 | 2.98 | 0 | 1 | 2 | 4 | 32 |
| | CountContract | 4.43 | 3.79 | 0 | 2 | 3 | 5 | 80 |
| | CountTotalFunction | 26.71 | 26.42 | 0 | 13 | 20 | 30 | 835 |
| | CountPublicVariable | 9.15 | 8.72 | 0 | 5 | 7 | 10 | 208 |
| | CountVariable | 12.64 | 11.32 | 0 | 7 | 9 | 14 | 212 |
| | CountFunctionPrivate | 0.59 | 2.37 | 0 | 0 | 0 | 0 | 64 |
| | CountFunctionInternal | 5.09 | 11.34 | 0 | 0 | 4 | 5 | 629 |
| | CountFunctionExternal | 1.86 | 7.04 | 0 | 0 | 0 | 1 | 280 |
| | CountFunctionPublic | 19.16 | 16.66 | 0 | 9 | 15 | 24 | 308 |
| Solidity-specific | NOI | 0.21 | 0.69 | 0 | 0 | 0 | 0 | 16 |
| | NOL | 0.63 | 0.92 | 0 | 0 | 1 | 1 | 27 |
| | NOSV | 16.31 | 21.11 | 0 | 7 | 10 | 17 | 1,367 |
| | NOMap | 3.40 | 3.10 | 0 | 3 | 3 | 4 | 89 |
| | NOPay | 1.12 | 1.87 | 0 | 0 | 1 | 1 | 33 |
| | NOE | 4.31 | 4.23 | 0 | 2 | 3 | 6 | 76 |
| | NOMod | 1.95 | 2.56 | 0 | 0 | 1 | 3 | 49 |
| | NOT | 1.30 | 2.03 | 0 | 0 | 1 | 2 | 56 |
| | NOC | 0.10 | 0.40 | 0 | 0 | 0 | 0 | 13 |
| | NODC | 0.00 | 0.12 | 0 | 0 | 0 | 0 | 16 |
| | NOSF | 9.51 | 11.50 | 0 | 4 | 8 | 11 | 395 |
| | SDFB | 0.51 | 0.50 | 0 | 0 | 1 | 1 | 1 |

$^{*}$:$Dim.$ refers to dimension; $Q2$ refers to median.

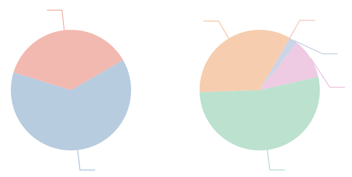


**Fig. 2:** The distribution of Smart Contract

Fig. 2 shows the distribution of crash-inducing smart contracts and non-crash-inducing smart contracts. We find that almost 63.1% of all smart contracts are non-crash-inducing ones (i.e., 34,539), while 36.9% of contracts are crash-inducing ones (i.e., 20,200). Among all crash-inducing smart contracts, we analyze the ratios of different types of runtime error in their transactions: 53.05% of “Reverted” (i.e., 1,177,412), 33.84% of “Out of Gas” (i.e., 750,944), 10.79% of “Bad Instruction” (i.e., 239,426), 2.32% of “Bad Jump Destination” (i.e., 51,525) and almost none of “Out of Stack” (i.e., 57).

## C. Experimental Setting

*1) **Baselines**:* We consider three state-of-the-art machine learning based methods (i.e., *SMARTEMBED* [37], *Safe* [38] and *ContractWard* [39]) and briefly introduce them as follows.

■ ***SMARTEMBED***. Gao et al. [37] proposed *SMARTEMBED* to detect bugs in smart contracts by code clone detection technology. More precisely, they build two datasets: the source code database containing all the verified smart contracts in the Etherscan and the bug database containing the bugs of each smart contract in its source code database. For a given smart contract, *SMARTEMBED* converts it into embedding matrices after some pre-processing operations, which is further used to conduct both clone-detection tasks and bug-detection tasks.

■ ***Safe***. Tann et al. [38] proposed an approach of sequential learning for defecting smart contract security threats. They adopt LSTM [40] to build the sequential security threats defection model with the input of operation codes (i.e., opcode) of smart contracts and their corresponding labels.

■ ***ContractWard***. Wang et al. [39] proposed a novel machine learning-based approach, *ContractWard*, to automatically detect vulnerabilities in smart contracts. *ContractWard* firstly convert source codes of smart contracts into operation codes (opcodes) and subsequently simplify the opcodes according to their designed rules. Then, *ContractWard* extracts thousands of dimensional bigram features from simplified contract opcodes to characteristic smart contracts and it is finally trained on a balanced dataset for vulnerability detection.

*2) **Performance Measures**:* We adopt two widely used performance metrics to evaluate CRASHSCDET's effectiveness.

*F1-measure*. There exist four possible prediction results for a smart contract when an approach classifies a smart contract as a crash-inducing one or not: True Positive(TP), False Positive(FP), False Negative(FN), and True Negative(TN). Therefore, based on above four possible results, F1-measure can be defined as follows: $F1\text{-}measure = \frac{2 \times Precision \times Recall}{Precision + Recall}$, where $Precision = \frac{TP}{TP+FP}$ and $Recall = \frac{TP}{TR+FN}$.

*AUC*. AUC [27, 41] represents the area under the receiver operating characteristic (ROC) curve [42], which is a 2D illustration of true positive rate (TPR) on the $y$-axis versus false positive rate (FPR) on the $x$-axis. ROC curve is obtained by varying the classification threshold over all possible values, separating clean and defect-prone predictions.

*3) **Validation Setting and Statistical Analysis**:* We consider the widely used 10-fold cross-validation setting and report the average of the 100 values. Besides, to check the statistical significance of the performance difference between the two different methods, we adopt the Wilcoxon signed-rank test [43] with a Bonferroni correction [44] at 95 percent significance level. Meanwhile, Cliff's delta ($\delta$) [45] is used to quantify the amount of difference between the two approaches.

## V. Empirical Study Results

### A. [RQ-1]: Can CRASHSCDET effectively identify crash-inducing smart contracts in advance?

**Motivation.** Smart contracts cannot be patched after deployed to the blockchain due to the immutability of the blockchain. Besides, smart contracts might contain a few specific defects compared to traditional programs (e.g., Android Apps) because of their unique and revolutionary features as compared to traditional software (e.g., the gas system, the decentralized characteristic). The transactions in history can be used to check the robustness, security, and high reliability of smart contracts, and all transactions of the smart contract are treated as test cases. However, the crash-inducing smart contracts (i.e., failed transactions of a smart contract) not only result in the consumption of funds (e.g., gas) but also affect the running efficiency of the Ethereum [3]. Therefore, detecting crash-inducing smart contracts in advance is a good way to ensure contacts' robustness.

**Method.** We first define 34 metrics to represent a smart contract, then we crawl all transactions of studied smart contracts to label whether they are executed successfully or not. Specially, we propose a machine learning-based method, CRASHSCDET, and implement it on top of the scikit-learn. By default, we build CRASHSCDET by using a Random Forest classifier and make a comparison with three state-of-the-arts (i.e., *SMARTEMBED* [37], *Safe* [38] and *ContractWard* [39]). Besides, we apply random oversampling on dataset to address imbalance issues with the help of *imblearn* package [46].

Additionally, Chen et al. [47] conducted a study and found that about 81% of accounts (96% of smart contracts and 77% of external owned account) have less than 5 transactions on Ethereum. That is, most accounts (especially smart contracts) are infrequent in transferring money. Besides, almost all smart contracts have no more than 30 transactions. Therefore, to remove low-quality contracts, we filter out two smaller datasets with two different numbers of transactions:5 and 30. Meanwhile, we remove duplicated contracts if the similarity of their bytecodes is 100%. Finally, we have a size of 25,276 for 5-trans dataset and a size of 13,954 for 30-trans dataset.

**Results.** Table III presents the average $F1\text{-}measure$ and $AUC$ values of four crash-inducing smart contract detectors. The top half of the table shows the results of 5-Trans. filter, while the bottom half of it shows the results on 30-Trans. filter. The statistical results are listed in the bottom few rows of each part, and the best model is shown in the last row of each part. According to the results shown in Table III, we find that the four crash-inducing smart contract detectors also have varying abilities to identify crash-inducing smart contracts and CRASHSCDET statistically significantly outperforms all baselines on both smaller datasets in terms of two performance measures with a large effect size in most cases.

More precisely, in terms of F1-measure on 5-Trans., CRASHSCDET achieves 0.791 on average and improves *Safe*(i.e., 0.644) by 22.8%. CRASHSCDET also performs similarly with both *ContractWard* (i.e, 0.789) and *SMARTEMBED* (i.e., 0.790). In terms of AUC on 5-Trans., CRASHSCDET obtains 0.870 on average and statistically significantly improves baselines by 33.0%-36.2% with a large effect size.

In terms of F1-measure on 30-Trans., CRASHSCDET performs similarly with *SMARTEMBED* and *ContractWard* on average, but markedly improves *Safe* by 60.4%. In terms of

**TABLE III:** The comparison results among three baselines and CrashSCDet

| | Classifer | F1-measure | | | | AUC | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | *CSCD* | *SE* | *Safe* | *CW* | *CSCD* | *SE* | *Safe* | *CW* |
| 5-Trans. | *Avg.* | 0.791 | 0.789 | 0.644 | 0.790 | 0.870 | 0.650 | 0.654 | 0.639 |
| | *Improv.* | | 0.3% | 22.8% | 0.1% | | 33.8% | 33.0% | 36.2% |
| | *p*-value | | <0.001 | <0.001 | <0.05 | | <0.001 | <0.001 | <0.001 |
| | Delta | | Small | Large | Negligible | | Large | Large | Large |
| | **Winner** | **CrashSCDet** | | | | **CrashSCDet** | | | |
| 30-Trans. | *Avg.* | 0.937 | 0.929 | 0.584 | 0.932 | 0.980 | 0.679 | 0.694 | 0.682 |
| | *Improv.* | | 0.9% | 60.4% | 0.5% | | 44.3% | 41.2% | 43.7% |
| | *p*-value | | <0.001 | <0.001 | <0.001 | | <0.001 | <0.001 | <0.001 |
| | Delta | | Large | Large | Large | | Large | Large | Large |
| | **Winner** | **CrashSCDet** | | | | **CrashSCDet** | | | |

∗ **CSCD** for CrashSCDet, **SE** for *SMARTEMBED*, **CW** for ContractWard, *Delta* for Cliff's Delta

AUC on 30-Trans., CrashSCDet obtains 0.980 on average and statistically significantly improves baselines by 41.2%-44.3% with a large effect size.

✍ **RQ-1** *Our proposed metrics can effectively capture the characteristics of the crash-inducing smart contracts and* CrashSCDet *can effectively identify the crash-inducing smart contracts and perform best among state-of-the-art approaches: SMARTEMBED, Safe and ContractWard.*

### *B.* [RQ-2]: How do different basic classifiers affect crash-inducing smart contract detector?

**Motivation**. In this paper, we propose a few metrics to characterize the smart contracts and build a crash-inducing smart contract detector with a basic classifier. In fact, different classifiers have different characteristics, which may have a varying affect on effectiveness of CrashSCDet.

**Method**. In order to indicate the influence of different basic classifiers on the effectiveness of CrashSCDet, we totally consider three classical and widely used basic classifiers (i.e., Random Forest(RF), Naive Bayes(NB), and Logistic Regression(LR)) in the scenario of software quality assurance [11, 13, 14, 17, 41, 48].

We build three CrashSCDets with different basic classifiers (i.e., RF, NB, and LR) on the top of scikit-learn toolkit with default parameters and re-conduct experiments on all original datasets. Meanwhile, we also apply random oversampling technology with the help of *imblearn* package to address the issues of data imbalance.

**TABLE IV:** The performance comparison among three CrashSCDets built with three different basic classifiers

| **Performance** | **F1-measure** | | | **AUC** | | |
|---|---|---|---|---|---|---|
| | RF | NB | LR | RF | NB | LR |
| *Average* | **0.75** | 0.40 | 0.59 | **0.83** | 0.64 | 0.67 |
| *Improvement* | | 87.5% | 27.1% | | 29.7% | 23.9% |
| *p-value* | | <0.001 | <0.001 | | <0.001 | <0.001 |
| *Cliff's Delta* | | Large | Large | | Large | Large |
| ***Winner*** | **CrashSCDet** $_{(RF)}$ | | | **CrashSCDet** $_{(RF)}$ | | |

**Results.** Table IV presents the average $F1\text{-}measure$ and $AUC$ values of the three crash-inducing smart contract detectors. The statistical results are shown in the bottom few rows of this table, and the best model is listed in the last row. According to the results shown in Table IV, we find that the three crash-inducing smart contract detectors have varying abilities to identify crash-inducing smart contracts. In particular, CrashSCDet $_{(RF)}$ statistically significantly outperforms CrashSCDet $_{(NB)}$ and CrashSCDet $_{(LR)}$ with a large effect size. On average, CrashSCDet $_{(RF)}$ achieves 0.75 of *F1-measure* and 0.83 of *AUC*, which improves CrashSCDet $_{(NB)}$ and CrashSCDet $_{(LR)}$ by 87.5% and 27.1% in terms of *F1-measure*, and by 29.7% and 23.9% in terms of *AUC*.

✍ **RQ-2** *Different basic classifiers (i.e., RF, NB and LR) have a varying affect on the effectiveness of* CrashSCDet *and* CrashSCDet *built with RF can achieve the best performance on identifying the crash-inducing smart contracts.*

### *C.* [RQ-3]: How do metrics in different dimensions affect crash-inducing smart contract detector?

**Motivation.** In addition to identifying crash-inducing smart contracts with high accuracy, we also are interested in investigating which types of metrics are the best contributors to the best detector. By default, our approach CrashSCDet combines the four dimensions of metrics: complexity, count, object-oriented, and Solidity-specific. These metrics characterize smart contracts from different aspects. Some aspects may be more discriminative for identifying crash-inducing smart contracts. Therefore, we want to figure out the importance of different dimensions of metrics.

**Method.** We build CrashSCDet by learning on metrics in each dimension. For the convenience of expression, we refer to them as the dimension name (i.e., "Complexity", "Count", "Object-oriented", and "Solidity-specific"). The CrashSCDet built with all metrics is referred to as "All". In each model, we keep the basic classifier (i.e., Random Forest with default settings) as the same. We compare their performance obtained by 10 times stratified 10-fold cross validation setting. In order to ensure a fair comparison, in each cross validation, we keep the training data and testing data the same for each model.

Moreover, to investigate whether the difference between the model built on all metrics and the four models built on a single dimension of metrics is statistically significant, we adopt the Wilcoxon signed-rank test with a Bonferroni correction at 95 percent significance level and compute the Cliff's delta to measure the effect size.

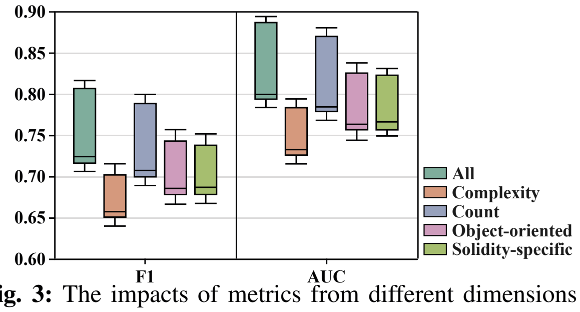


**Fig. 3:** The impacts of metrics from different dimensions on the performance of CrashSCDet

**Results.** Fig. 3 illustrates the performance of five models in terms of *F1-measure* and *AUC* in the form of the boxplot. From this figure, we find that all models can achieve a good performance and "All" performs the best, while "Complexity" performs the worst. Among the four models built on each dimension (i.e., "Complexity", "Count", "Object-oriented" and "Solidity-specific"), "Count" performs the best, while "Object-oriented" and "Solidity-specific" perform similarly. In particular, "All" achieves 0.75 of *F1-measure* and 0.83 of *AUC*, which statistically significantly outperform "Complexity" (0.67 of *F1* and 0.74 of *AUC*), "Count" (0.73 of *F1* and 0.81 of *AUC*), "Object-oriented" ( 0.70 of *F1* and 0.78 of *AUC*), and "Solidity-specific" (0.70 of *F1* and 0.78 of *AUC*).

✍ **RQ-3** *Metrics in different dimensions have varying abilities to depict the characteristic of smart contracts and "Count" is the most discriminative dimension. Combining all the metrics can improve the performance of a crash-inducing smart contract detector.*

## VI. Threats to Validity

Threats to internal validity mainly consist in the validation of studied smart contracts. We collect the studied smart contracts on July 31, 2019. However, the Ethereum ecosystem is fast-evolving and some smart contracts may be destructed. Besides, we label a smart contract as crashed one or non-crashed one based on the traces of the transactions on October 10, 2020. As the running of smart contracts, the traces of transactions of smart contracts may also be changed.

Threats to external validity mainly consist in the generalizability of our approach. We only consider the verified smart contracts deployed in Etherscan. Thus, it is still unknown whether our conclusions are generalizable to those contracts un-deployed on Ethereum.

Threats to construct validity mainly lie in the adopted performance metrics. We consider two widely used performance measures (e.g., *F1-measure* and *AUC*) and check the approach's superiority with Wilcoxon signed-rank test.

## VII. Related Work

Ethereum, invented by Vitalik et al. [49] at the end of 2015, introduces a new technology named smart contracts to the world. It is a blockchain-based cryptocurrency system. Compared with Bitcoin, Ethereum defines a Turing-complete programming platform and a run-time environment called EVM (Ethereum Virtual Machine). EVM can run the bytecodes of smart contracts. Nowadays, Ethereum has become the most popular platform on which to run smart contracts and many work related to smart contracts have been proposed [19, 50].

As smart contracts provide programmers with a new platform, many new tools are proposed to help to perform code analysis and verification. Tonelli et al. [51] firstly defined some metrics which are similar to those of the classic Chidamber&Kemerer (C&K) [24] metrics in the object-oriented world. Then, they compared their distributions with the metrics extracted from more traditional software projects on more than twelve thousands smart contracts written in Solidity. The results indicated that, in general, the ranges of smart contracts metrics are more restricted than the corresponding metrics of traditional software systems. Following that, Peter [52] further defined and extracted C&K software metrics and also made a deep analysis on the distributions of metrics on 40,352 smart contracts from various dimensions.

Jiang et al. [4] proposed a fuzzing-based tool named *ContractFuzzer* to detect seven types of security defects. Nguyen et al. [5] proposed a method named *sFuzz* to identify more security defects by covering more branches. Kolluri et al. [53] proposed a method named *EthRacer* which can run directly on Ethereum bytecode. Liu et al. [54] proposed a method named *ReGuard*, which can provide a web service for developers to easily use. Fu et al. [55] proposed a method named *EVMFuzz* which designs a differential fuzz testing framework and supports EVM smart contracts developed by different programming languages.

Besides, Liu et al. [10] proposed a novel semantic-aware security auditing technique called S-gram for Ethereum, which can be used to predict potential vulnerabilities by identifying irregular token sequences. Chang et al. [56] propose an new approach, sCompile, to automatically identify critical program paths in a smart contract, sort paths according to their criticality, discard them if they are not feasible or otherwise provide them with user-friendly warnings for user inspection. Gao et al. [37] use clone detection method to propose a tool named *SMARTEMBED* to detect bugs in smart contracts.

Prior work either needs to exhaust a large amount of gas or detects a few specific types of defects. Different from previous work, in this paper, we mainly focus on identifying crashed smart contract which has runtime errors. That is, this paper proposes CrashSCDet to detect whether the contract will crash, rather than only focusing on specific defects.

## VIII. Conclusions

In this paper, we mainly focus on whether static source code metrics can help identify crash-inducing smart contracts in advance or not. We firstly propose 34 static source code metrics from four dimensions (i.e., *complexity metrics*, *count metrics*, *object-oriented metrics* and *Solidity-specific metrics*) to characterize a smart contract. Then, we totally collect 54,739 verified smart contracts source codes on Ethereum and extract these static metrics of each contract. We label these contracts with their corresponding transactions. Following that, we propose a crash-inducing smart contract detector named CrashSCDet, which can effectively identify whether a smart contract is a crash-inducing one. We further analyze the importance of metrics from different dimensions and find that "count metric" is the most important one.

## Acknowledgment

This research is supported by the National Key Research and Development Program of China (No. 2021YFB2701100), the National Natural Science Foundation of China (No. 62202419), the Ningbo Natural Science Foundation (No. 2022J184), the National Science Foundation of China (No. U20A20173), the Meituan Company, the Fundamental Research Funds for the Central Universities (No. 226-2022-00064) and Singapore under its Industry Alignment Fund – Pre-positioning (IAF-PP) Funding Initiative.

## References

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Manubot, Tech. Rep., 2019.

[2] Ethereum, 2022. [Online]. Available: https://ethereum.org/

[3] S. Jumnongsaksub and K. Sripanidkulchai, "Reducing smart contract runtime errors on ethereum," *IEEE Software*, 2020.

[4] B. Jiang, Y. Liu, and W. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 259–269.

[5] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," *arXiv preprint arXiv:2004.08563*, 2020.

[6] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[7] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[8] J. Chen, X. Xia, D. Lo, J. Grundy, X. Luo, and T. Chen, "Defectchecker: Automated smart contract defect detection by analyzing evm bytecode," *IEEE Transactions on Software Engineering*, 2021.

[9] J. Chen, X. Xia, D. Lo, J. Grundy, and X. Yang, "Maintaining smart contracts on ethereum: Issues, techniques, and future challenges," *arXiv preprint arXiv:2007.00286*, 2020.

[10] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: towards semantic-aware security auditing for ethereum smart contracts," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 814–819.

[11] C. Ni, X. Xia, D. Lo, X. Chen, and Q. Gu, "Revisiting supervised and unsupervised methods for effort-aware cross-project defect prediction," *IEEE Transactions on Software Engineering*, 2020.

[12] B. Turhan, T. Menzies, A. B. Bener, and J. Di Stefano, "On the relative value of cross-company and within-company data for defect prediction," *Empirical Software Engineering*, vol. 14, no. 5, pp. 540–578, 2009.

[13] Y. Zhou, Y. Yang, H. Lu, L. Chen, Y. Li, Y. Zhao, J. Qian, and B. Xu, "How far we have progressed in the journey? an examination of cross-project defect prediction," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 1, pp. 1:1–1:51, 2018.

[14] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE transactions on software engineering*, no. 1, pp. 2–13, 2007.

[15] M. H. Halstead *et al.*, *Elements of software science*. Elsevier New York, 1977, vol. 7.

[16] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, pp. 308–320, 1976.

[17] S. Herbold, A. Trautsch, and J. Grabowski, "A comparative study to benchmark cross-project defect prediction approaches," *IEEE Trans. Software Eng.*, vol. 44, no. 9, pp. 811–833, 2018.

[18] C. Ni, "Replication package," 2022. [Online]. Available: https://github.com/jacknichao/CrashSCDet

[19] T. Chen, Y. Zhu, Z. Li, J. Chen, X. Li, X. Luo, X. Lin, and X. Zhange, "Understanding ethereum via graph analysis," in *IEEE INFOCOM 2018-IEEE conference on computer communications*. IEEE, 2018, pp. 1484–1492.

[20] Online, "Ethereum virtual machine opcodes," 2021. [Online]. Available: https://ethervm.io/

[21] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[22] scikit learn, 2022. [Online]. Available: https://scikit-learn.org/stable/

[23] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[24] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[25] T. M. Meyers and D. Binkley, "An empirical study of slice-based cohesion and coupling metrics," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 1, pp. 1–27, 2007.

[26] T. Zimmermann and N. Nagappan, "Predicting subsystem failures using dependency graph complexities," in *Software Reliability, 2007. ISSRE '07. The 18th IEEE International Symposium on*, 2007.

[27] C. Ni, W.-S. Liu, X. Chen, Q. Gu, D.-X. Chen, and Q.-G. Huang, "A cluster based feature selection method for cross-project software defect prediction," *Journal of Computer Science and Technology*, vol. 32, no. 6, pp. 1090–1107, 2017.

[28] py-solc x, 2022. [Online]. Available: https://github.com/iamdefinitelyahuman/py-solc-x

[29] py-solc ast, 2022. [Online]. Available: https://github.com/iamdefinitelyahuman/py-solc-ast

[30] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," *IEEE Transactions on Software engineering*, vol. 31, no. 10, pp. 897–910, 2005.

[31] R. Martin, "Oo design quality metrics," *An analysis of dependencies*, vol. 12, no. 1, pp. 151–170, 1994.

[32] P. Bhattacharya and I. Neamtiu, "Assessing programming language impact on development and maintenance: A study on c and c++," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 171–180.

[33] M. Garkavtsev, N. Lamonova, and A. Gostev, "Chosing a programming language for a new project from a code quality perspective," in *2018 IEEE Second Interna-*

*tional Conference on Data Stream Mining & Processing (DSMP)*. IEEE, 2018, pp. 75–78.

[34] D. Gupta, "What is a good first programming language?" *Crossroads*, vol. 10, no. 4, pp. 7–7, 2004.

[35] Solidity, 2022. [Online]. Available: https://github.com/ethereum/solidity

[36] EtherScan, July 31, 2019. [Online]. Available: Ethereumblockchainexplorersite.https://etherscan.io/

[37] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Smartembed: A tool for clone and bug detection in smart contracts through structural code embedding," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 394–397.

[38] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, "Towards safer smart contracts: A sequence learning approach to detecting security threats," *arXiv preprint arXiv:1811.06632*, 2018.

[39] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, 2020.

[40] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[41] S. Herbold, A. Trautsch, and J. Grabowski, "Global vs. local models for cross-project defect prediction," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1866–1902, 2017.

[42] J. A. Hanley and B. J. McNeil, "The meaning and use of the area under a receiver operating characteristic (roc) curve." *Radiology*, vol. 143, no. 1, pp. 29–36, 1982.

[43] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[44] H. Abdi, "Bonferroni and šidák corrections for multiple comparisons," *Encyclopedia of measurement and statistics*, vol. 3, pp. 103–107, 2007.

[45] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[46] Imbalance, 2022. [Online]. Available: https://pypi.org/project/imblearn/

[47] 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.

[48] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, "A large-scale empirical study of just-in-time quality assurance," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, 2013.

[49] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, 2014.

[50] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, 2020.

[51] R. Tonelli, G. Destefanis, M. Marchesi, and M. Ortu, "Smart contracts software metrics: a first study," *arXiv preprint arXiv:1802.01517*, 2018.

[52] P. Hegedűs, "Towards analyzing the complexity landscape of solidity based ethereum smart contracts," *Technologies*, vol. 7, no. 1, p. 6, 2019.

[53] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 363–373.

[54] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 65–68.

[55] Y. Fu, M. Ren, F. Ma, Y. Jiang, H. Shi, and J. Sun, "Evmfuzz: Differential fuzz testing of ethereum virtual machine," *arXiv preprint arXiv:1903.08483*, 2019.

[56] 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.