2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678597

Characterizing Transaction-Reverting Statements in Ethereum Smart Contracts

Lu Liu[a,b], Lili Wei[b], Wuqi Zhang[b], Ming Wen[c], Yepang Liu[a*], Shing-Chi Cheung[b*]
[a] Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China
[b] Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China
[c] School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, China
{lliubf, liliwei, wzhangcb, scc}@cse.ust.hk, {mwenaa}@hust.edu.cn, {liuyp1}@sustech.edu.cn

***Abstract*—Smart contracts are programs stored on blockchains to execute transactions. When input constraints or security properties are violated at runtime, the transaction being executed by a smart contract needs to be reverted to avoid undesirable consequences. On Ethereum, the most popular blockchain that supports smart contracts, developers can choose among three transaction-reverting statements (i.e., `require`, `if...revert`, and `if...throw`) to handle anomalous transactions. While these transaction-reverting statements are vital for preventing smart contracts from exhibiting abnormal behaviors or suffering malicious attacks, there is limited understanding of how they are used in practice. In this work, we perform the first empirical study to characterize transaction-reverting statements in Ethereum smart contracts. We measured the prevalence of these statements in 3,866 verified smart contracts from popular dapps and built a taxonomy of their purposes via manually analyzing 557 transaction-reverting statements. We also compared template contracts and their corresponding custom contracts to understand how developers customize the use of transaction-reverting statements. Finally, we analyzed the security impact of transaction-reverting statements by removing them from smart contracts and comparing the mutated contracts against the original ones. Our study led to important findings. For example, we found that transaction-reverting statements are commonly used to perform seven types of authority verifications or validity checks, and missing such statements may compromise the security of smart contracts. We also found that current smart contract security analyzers cannot effectively handle transaction-reverting statements when detecting security vulnerabilities. Our findings can shed light on further research in the broad area of smart contract quality assurance and provide practical guidance to smart contract developers on the appropriate use of transaction-reverting statements.**

***Index Terms*—Ethereum, smart contract, transaction-reverting statement, empirical study, security vulnerability**

I. Introduction

Smart contracts are programs stored on blockchains to execute transactions. In recent years, smart contracts have been widely used for various purposes such as to offer financial services [1]. Ethereum [2] is the largest decentralized platform for smart contracts with the second biggest blockchain market capitalization [3]. There are over one million transactions executed on Ethereum daily [4].

As smart contracts are often used to manage valuable user assets, their security is of paramount importance. Anomalous transactions caused by various runtime errors should be detected and reverted promptly to prevent undesirable consequences such as financial losses. In Solidity [5], the most popular programming language for Ethereum smart contracts, there are three statements that can help detect runtime errors and revert transactions, namely, `require`, `if...revert`, and `if...throw`. Figure 1 shows the example uses of these *transaction-reverting statements* to revert transactions submitted by unauthorized senders. While all three statements can revert transactions when anomalous conditions occur, the first two would refund the unused gas to transaction senders.

```
if (msg.sender != owner) {revert(); }
require(msg.sender == owner);
if (msg.sender != owner) {throw; }
```

Fig. 1. Examples of transaction-reverting statements

Transaction-reverting statements are frequently used in smart contracts. Our analysis reveals that over 94% of smart contracts use transaction-reverting statements in certain ways. Surprisingly, this figure is even higher than that of general-purpose `if` statements. These statements are also frequently discussed in the Solidity developers community. We searched on Stack Overflow [6], the most popular Q&A website for programmers, using the keywords "require()", "revert()", and "if throw" under the tag "solidity". As of August 2021, there are already 1,280 questions related to the three transaction-reverting statements, many of which have been viewed thousands of times.

Transaction-reverting statements can effectively help prevent smart contracts from exhibiting abnormal behaviors or suffering malicious attacks. For example, in the SWC Registry [7], which indexes common smart contract weaknesses, there is a kind of weakness called "Unchecked Call Return Value" (SWC-104 [8]). This weakness occurs when the return value of a message call is not properly checked in a smart contract. To ease understanding, we give an illustrative example in Figure 2. In the code snippet, the `callNotChecked()` function does not check the return value of `callee.call()` (Line 2). When the execution of `callee.call()` fails, the `callNotChecked()` function would not do anything. This may cause serious and irreversible consequences, e.g., the contract announces to the caller with error execution information that the call has been executed

* Yepang Liu and Shing-Chi Cheung are the corresponding authors.

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00062

```
1 function callNotChecked(address callee) public {
2     callee.call();
3 }
4
5 function callChecked(address callee) public {
6     require(callee.call());
7 }
```

Fig. 2. An example of the *Unchecked Call Return Value* weakness

successfully, but actually, the call fails. To fix the weakness, developers are suggested to add a `require` statement to check the execution status of `callee.call()` (such as in Line 6 of `callChecked()`) so that the anomalous transactions can be reverted and the unused gas can be returned to the transaction sender upon unsuccessful execution of `callee.call()`.

As we can see from the above example, appropriate uses of transaction-reverting statements can help improve the reliability and security of smart contracts. However, there is little research on transaction-reverting statements. Without a comprehensive understanding of how these statements are used in practice, one cannot design tools to effectively identify the inappropriate uses of such statements or formulate good practices to help smart contract developers. We conducted the first empirical study to characterize transaction-reverting statements in Ethereum smart contracts to bridge the gap. Specifically, we investigated the following four research questions:

- **RQ1 (Prevalence):** *Are transaction-reverting statements commonly used in Ethereum smart contracts?*
- **RQ2 (Purpose):** *What are the major purposes of using transaction-reverting statements in smart contracts?*
- **RQ3 (Developer Customization):** *Are there differences between template contracts and custom contracts in terms of using transaction-reverting statements?*
- **RQ4 (Security Impact):** *Are there any security consequences if transaction-reverting statements are missing in smart contracts?*

For the study, we constructed a dataset of 270 template contracts and 3,866 dapp contracts, which were collected from popular template code repositories [9]–[12] and real-world dapps with millions of transactions. To answer RQ1, we measured the code density of transaction-reverting statements and compared it with that of general-purpose `if` statements in smart contracts. To answer RQ2, we built a taxonomy of the purposes of transaction-reverting statements via an inductive coding process [13]. To answer RQ3, we leveraged a code clone detector to identify contracts developers customized from popular-used contract templates and studied how developers customize transaction-reverting statements at a fine granularity of clauses of conditions based on template contracts. To answer RQ4, we analyzed the security impact of transaction-reverting statements by removing them from smart contracts and comparing the mutated contracts against the original ones. Our major findings include:

- Over 94% of our analyzed smart contracts use transaction-reverting statements. Comparatively, only 87.9% of them use general-purpose `if` statements. This shows that transaction-reverting statements are pervasively used in real-world smart contracts and may play important roles in assuring the correct execution of transactions.
- Transaction-reverting statements are commonly used to perform seven types of security-critical checks, such as verifying user authorities.
- Developers are most likely to strengthen transaction-reverting statements by adding clauses, variables, or new transaction-reverting statements. The customized transaction-reverting statements are commonly used for range checks and logic checks.
- The lack of transaction-reverting statements may introduce security issues to smart contracts. Existing smart contract security analyzers show weak support in handling transaction-reverting statements when detecting security vulnerabilities.

To summarize, the main contribution of this work is the character study of transaction-reverting statements in Ethereum smart contracts. To the best of our knowledge, this study is the first of its kind. The findings can facilitate further research in smart contract quality assurance and provide practical guidance to smart contract developers on the appropriate use of transaction-reverting statements. Our data are released on GitHub for public usage [14].

The organization of the remaining sections is as follows. In Section II, we introduce some related background knowledge. Section III presents how we constructed four datasets of smart contracts for empirical analysis. Then in Section IV, we present the design of the empirical study to answer the four research questions and introduce our data analysis methodologies and empirical findings. We discuss threats to the validity of our studies in Section V. After that, we discuss related works in Section VI and conclude our work in Section VII.

II. Background

This section presents the background and explains the terminologies used in the paper.

A. Smart Contracts & Dapps

Smart contracts are autonomous programs running on blockchains like Ethereum [2]. The execution of smart contracts does not rely on a trusted third party and is fully decentralized. Dapps are decentralized applications that can offer end-users various functionalities. The core logic of dapps is backed by smart contracts to meet the requirements of applications. Solidity [5] is the most popular high-level language to program Ethereum smart contracts. In this paper, we focus on the smart contracts written in Solidity.

B. Error-Handling Statements

Solidity uses state-reverting exceptions to handle errors. It provides four statements to deal with errors, namely, `require`, `if...revert`, `assert`, and `if...throw`. If

these statements identify the occurrence of erroneous conditions, they will throw an exception and revert the blockchain and contract state to the state before the execution of the transaction. The four error-handling statements can be further divided into the following two categories [5]:

- **Transaction-reverting statements** refer to the `require`, `if...revert`, and `if...throw` statements, that are used to check for erroneous conditions. Before version 0.4.10, Solidity provides the `if...throw` statement for reverting transactions. As the language evolves, there are two more alternatives, namely, `require` and `if...revert`, to replace `if...throw` since Solidity 0.4.10. The `if...throw` statement is officially deprecated in Solidity 0.4.13. These statements can all trigger state reversion when erroneous conditions occur. The only difference between `if...throw` and the two replacements is that `if...throw` will use up all remaining gas when errors occur, while the two replacements will refund the remaining gas to the transaction sender.
- **The assertion statement** `assert` should only be used for debugging purpose, which are not supposed to exist in production code. If a specified assertion is violated, it means that the contract has a bug, which needs to be fixed.

In our study, we focus on transaction-reverting statements. Since `if...throw` statement is already deprecated, we mainly investigate the use of `require` and `if...revert` statements in real-world smart contracts. In the following of this paper, transaction-reverting statements refer to `require` and `if...revert` statements if not otherwise specified.

C. Template Contracts & Custom Contracts

Writing a smart contract is non-trivial for developers, especially when there is a high demand for security [15]. To facilitate contract development and prevent vulnerabilities, *template contracts* are provided by industrial institutions and organizations for different use cases. These template contracts are usually well-maintained and provide many high-quality or fully functional components for reuse. In practice, many developers copy or reuse components in template contracts in their own contracts, which we call *custom contracts*, to save efforts and ensure security. Developers' customizations may add, delete, or modify existing transaction-reverting statements for various purposes.

D. Solidity Components

Smart contracts written in Solidity are put in `.sol` files, each of which may contain one or more components of three kinds: *contracts*, *libraries*, and *interfaces*. Template contract codebases often provide a set of such Solidity components that developers can reuse.

III. Dataset Construction

To investigate our research questions, we constructed four datasets of smart contracts for empirical analysis. This section explains how these datasets were constructed.

TABLE I
Information of The Dapp Contract Dataset

Category	# Contracts	Total LOC	Avg LOC	# Transactions
Exchanges	371	256,652	695.53	1,001,698
Finance	621	460,218	742.29	1,257,473
Gambling	844	864,530	1,018.29	712,299
Game	1,152	853,524	708.32	2,028,069
High-risk	446	265,960	593.66	782,583
Marketplaces	140	120,620	815.00	395,204
Social	70	45,146	654.29	152,429
Utilities	132	78,737	601.05	376,976
Others	90	55,950	608.15	80,302
Total	3,866	3,001,337	763.50	6,787,033

A. Crawling Dapp Contracts

As of April 2021, over 40 million smart contracts have been deployed on Ethereum [16]. Despite the large volume, many Ethereum smart contracts are deprecated or rarely used (with few transactions). In our empirical study, we aim to analyze representative smart contracts that are often used in real life. For this purpose, we chose to collect smart contracts from popular dapps. Such collected smart contracts are of higher quality, more frequently used, and better maintained.

Specifically, we collected smart contracts from all 1,699 dapps indexed by Dapp.com [17], a popular dapp collection website, in February 2021 by referring to the contract addresses listed in the description of dapps. We found that most dapps have less than 200 contracts, but the dapp Uniswap is an exception. Uniswap [18] is a decentralized exchange that allows users to exchange one kind of token for another kind. It has 3,964 smart contracts because there is a contract factory, which will create a contract for every directly exchangeable token pair on Uniswap, and most such created contracts share the same code. To reduce the impact of data imbalance, we randomly selected 200 contracts for Uniswap (i.e., downsampling). For the other dapps, we collected all the addresses of their used smart contracts listed on Dapp.com. Then, we leveraged the APIs provided by Etherscan [19], an Ethereum block explorer, to collect contract source codes. In total, we collected 6,016 smart contracts, and 3,866 of them are verified ones with source code available, which will be used in the subsequent studies. Table I provides the demographic information of the 3,866 verified contracts. As we can see, they are from different categories, contain hundreds of lines of code (on average), and have a large number of transactions.

B. Collecting Template Contracts

Template contracts play an important role in the Ethereum ecosystem. When reusing them, developers may customize the transaction-reverting statements. To study such customizations, we built a dataset of custom contracts and the corresponding template contracts. For template contracts, we collected them from four data sources, which contain smart contracts that are widely used on Ethereum. For custom contracts, we explain how to identify them in the next subsection. Table II shows the popularity of the data sources of template contracts, and we introduce each of them in the following.

TABLE II
THE POPULARITY OF TEMPLATE CONTRACT DATA SOURCES

Data Source	# GitHub Stars	# Repository Forks
OpenZeppelin	10k	4.5k
aragonOS	488	190
ConsenSys	4.1k	774
EIPs	6.2k	2.4k

- OpenZeppelin [9] is a library for secure smart contract development, which provides reusable contract templates such as implementations of token standards to help build custom contracts. We collected 115 contracts from OpenZeppelin.
- aragonOS [10] is a smart contract framework for building decentralized organizations, dapps, and protocols. We collected 107 contracts from aragonOS.
- ConsenSys [11] provides Solidity smart contract code for simple, standards-compliant tokens on Ethereum. We collected 34 contracts from ConsenSys.
- Besides the above data sources, we also collected 14 final EIPs (Ethereum Improvement Proposals) with 10 reusable template contracts from the ERC website [12].

In total, we collected 270 template contracts.

C. Identifying Custom Contracts

It isn't easy to associate template contracts with custom contracts because developers rarely explicitly specify the templates they reuse to write smart contracts. To identify custom contracts, we leveraged a code clone detection tool, SmartEmbed [20], to calculate the code similarity between the 270 template contracts and our collected 3,866 dapp contracts. If the code similarity between a dapp contract and a template contract is higher than or equal to 85%, we consider that the dapp contract is a custom contract based on the template contract. Via this process, we identified a set of 227 custom contracts based on 74 template contracts. We give more details of the custom contract dataset in Section IV-C.

D. Creating Mutated Contracts

To investigate the security impact of transaction-reverting statements in smart contracts, we constructed a dataset of mutated contracts by removing all transaction-reverting statements in the 3,866 contracts. The mutated contracts were later analyzed by existing smart contract vulnerability detection tools to assess their security. A detailed description of the mutated contracts is given in Section IV-D.

IV. EMPIRICAL STUDY

With the four datasets, we conducted a large-scale empirical study, aiming to 1) understand the use of transaction-reverting statements in smart contracts, 2) identify good/bad practices, and provide suggestions to help developers appropriately use transaction-reverting statements, and 3) inspire future research. In this section, we present our data analysis methodology and empirical findings for each of the four research questions listed in Section I.

TABLE III
CODE DENSITY OF CONDITIONAL STATEMENTS

Conditional Statement Type	Total Lines of Statements	Code Density
Transaction-reverting Statement	67,770	49.76
`if...throw` Statement	2,061	286.74
General-purpose `if` Statement	66,122	86.92

A. RQ1 (Prevalence)

Study Methodology: To answer RQ1, we measured the prevalence of transaction-reverting statements in smart contracts. Specifically, we first identified all the transaction-reverting statements in the 3,866 dapp contracts and then computed the code density of these statements. Following existing practices [21], [22], we computed code density for transaction-reverting statements as LOC/LOT, where LOC is the lines of code of a contract and LOT is the lines of transaction-reverting statements. Similarly, we also computed the code density for general-purpose `if` statements and `if...throw` statements for comparison. Note that we separately analyzed general-purpose `if`, `if..throw`, and `if...revert` statements. When an `if` statement is used with `throw` or `revert`, we will not consider it as a general-purpose `if` statement since it is used to revert transactions. In addition, we counted the number of transaction-reverting statements within a `if...throw` or `if...revert` code block as one.

Finding 1: *In our analyzed smart contracts, transaction-reverting statements are more frequently used than general-purpose `if` statements.*

Among all the 3,866 contracts, 3,647 (94.3%) contracts contain transaction-reverting statements, while only 3,399 (87.9%) contracts contain general-purpose `if` statements. Table III gives the detailed results, where the column *"Total Lines of Statements"* lists the total number of the concerned statements in the whole dataset and the *"Code Density"* column shows the average code density per contract for each type of statement. As shown in the table, transaction-reverting statements are more frequently used than general-purpose `if` statements in terms of both metrics. On average, there is one transaction-reverting statement per 49.76 lines of code, while general-purpose `if` statements are used once per 86.92 lines.

Finding 2: *8.6% of our analyzed smart contracts are still using the deprecated `if...throw` statements, which may cause unnecessary financial loss to users.*

As explained in Section II-B, `if..throw` statements can also help revert transactions but using them would incur additional costs of gas and induce unnecessary financial loss to the contract users. As a result, `require` and `if...revert` statements were introduced in Solidity 0.4.10 as replacements and `if..throw` was officially deprecated since Solidity 0.4.13 in 2017. However, we found that 332 (8.6%) of our analyzed smart contracts are still using `if...throw` statements. Besides, in 252 smart contracts (6.5%), there exists a mixed use of `if...throw` and `require` statements. We

further collected the Solidity versions used in these 3,866 contracts. Our results showed that 43 contracts (1.1%) still use Solidity versions before 0.4.10. Such contracts can only use the deprecated `if...throw` statements to revert transactions. The users who submit transactions to these contracts may suffer from unnecessary costs of gas.

Answer to RQ1: *Transaction-reverting statements are more frequently used in smart contracts than general-purpose `if` statements. There are still a non-negligible proportion of contracts using deprecated `if..throw` statements, which may incur unnecessary gas consumption when transactions revert.*

Implication: *Transaction-reverting statements may play an essential role in assuring the correct execution of transactions. Researchers working on smart contract quality assurance and security analysis should pay more attention to such statements as inappropriately using them may lead to abnormal contract behaviors or financial losses.*

B. RQ2 (Purpose)

Study Methodology: To understand the purposes of using transaction-reverting statements, we manually analyzed our collected smart contracts with the following two steps:

Step 1: Statement selection. Since there are 67,770 transaction-reverting statements in the 3,866 dapp contracts, it is infeasible to analyze all of them manually. For our study, we randomly selected 382 of these statements, representing the whole set with a confidence level of 95% and a confidence interval of 5%. For the 270 template contracts, we analyzed all 175 transaction-reverting statements in them.

Step 2: Constructing the purpose taxonomy. To understand and categorize the purposes, we first sampled 100 of the 557 (= 382 + 175) transaction-reverting statements for a pilot construction of the taxonomy. Similar to many existing empirical studies, we followed an open coding procedure [13] to inductively create the categories of our taxonomy in a bottom-up manner. Two authors read all the sampled transaction-reverting statements and the corresponding contracts to understand their purposes. The two authors also considered the string arguments of the transaction-reverting statements provided by the contract owners and the comments around the transaction-reverting statements when comprehending the contract code. They categorized the 100 statements independently and marked those unclear or insufficient categories. They then discussed and adjusted their category tags during meetings with the help of a third author to resolve conflicts. In this way, we successfully constructed the pilot taxonomy.

Based on the coding schema in the pilot taxonomy, the two authors continued to label the remaining 457 transaction-reverting statements for two more iterations. In these two iterations, the two authors went back and forth between categories and transaction-reverting statements to refine the taxonomy. The conflicts of labeling were again discussed during meetings and resolved by the third author. In this way, we adjusted the pilot taxonomy and obtained the final results. We used the Cohen's Kappa score [23] to measure the agreement between the two authors. The overall score is 0.73, indicating that the two authors had a high agreement on the taxonomy.

As shown in Table IV, the final taxonomy is organized into two categories, each of which is further divided into sub-categories. There is no overlap between these sub-categories, i.e., a clause in a transaction-reverting statement can only be classified into one of them. The table also provides illustrative examples collected from our datasets to ease understanding.

Finding 3: *Transaction-reverting statements are commonly used to perform seven types of authority verifications or validity checks.*

Authority Verification. 76 of the 435 clauses in the 382 transaction-reverting statements in the dapp contracts are for *Authority Verification*. The figure for the template contracts is 31 of 175. Authority Verification aims to check whether a given contract address or token ID is authorized by the contract owner for the sake of security:

- ***Address Authority Check*** is to check whether a given address, mostly the address of the transaction sender, is authorized by the contract owner. We observed two types of address checks. One is to check whether the given address equals to a specified address. The other is to check whether the given address is within a list of authorized addresses. The proportions of transaction-reverting statements that perform address authority checks in dapp contracts and template contracts are 14.9% and 16.0%, respectively.
- ***Token Verification***. Tokens are value counters stored in a contract, which are mappings of addresses to account balances. Token verification checks whether a given token ID is authorized, i.e., within the mappings of addresses. The proportions of transaction-reverting statements that perform token verification in dapp contracts and template contracts are 2.5% and 1.7%, respectively.

Validity Check. 359 of the 435 clauses in the 382 transaction-reverting statements in the dapp contracts are for validity checks. The figure for the template contracts is 144 of 175. Generally, validity checks are performed to check if certain runtime values are valid, i.e., satisfying pre-defined conditions. We observed five sub-categories of validity checks:

- ***Logic Check*** refers to the use of logical operators to check the validity of certain runtime values. Such checks are commonly seen in the conditions of transaction-reverting statements, such as checking the return value of a low-level function call, checking the value of a boolean flag, and so on. 37.7% dapp contracts and 29.1% template contracts contain transaction-reverting statements for logic checks.
- ***Range Check*** is to check whether a runtime value (e.g., an input) is within a specific range. 29.4% dapp contracts and 26.9% template contracts contain transaction-reverting statements for range checks.
- **Overflow/Underflow Check** is to check whether an input value crosses the limit of the prescribed size for a data type. 14.9% template contracts contain transaction-reverting

TABLE IV
THE PURPOSES OF USING TRANSACTION-REVERTING STATEMENTS IN THE PARAGRAPHS.

Category	First-Level Sub-Category	Second-Level Sub-Category	Description	Illustrative Example
Authority Verification	Address Authority Check	Equal to a specific address	Check whether a contract address is equal to a 20 bytes address specified by the contract owner.	`require(msg.sender == address(nonFungibleContract));`
		Within a specific address list	Check whether a contract address is in a address list provided by the contract owner.	`require(isAuthorized(msg.sender, msg.sig));`
	Token Verification	-	Some developers use a token ID to identify a contract address. The verification of the token ID is actually the verification of a contract address.	`require(_exists(tokenId), "URI query for nonexistent token");` (*_exists()* maps a token ID to a contract address using a mapping to verify the contract address.)
Validity Check	Logic Check	-	Check runtime values using logical operators.	`require(!_mintingFinished);`
	Range Check	-	Check whether the value of a variable is within a value range.	`require(underlyingBalance > 0, "Not have any liquidity deposit");`
	Overflow/ Underflow Check	-	Check whether the value of the variable is out of range of the declared data type.	`require((z = x + y)>= x);`
	Arithmetic Check	-	Check runtime values against arithmetic constraints.	`require(b != 0); c = a / b;`
	Address Validity Check	-	Check whether a contract address is a valid one.	`require(owner != address(0));`

statements for overflow/underflow checks, while the ratio is only 3.9% for dapp contracts. We further investigated the corresponding template contracts and found that many of them adopt the `SafeMath` library, which provides safe number operations to protect contracts from overflow/underflow vulnerabilities. This also shows that template contracts emphasize more on security, comparing than ordinary dapp contracts.

- ***Arithmetic Check*** is to check whether the value of a variable violates common constraints in arithmetic operations, such as divided by 0, mod 0, etc. These checks are not frequently performed comparing to the above categories. Only 0.7% dapp contracts and 2.3% template contracts contain transaction-reverting statements for *arithmetic checks*.
- ***Address Validity Check*** is to check whether a contract address is valid. Note that this is different from the address authority check discussed above, which is to check whether an address is an authorized one (a valid address may not be authorized). For example, a common address validity check is to check whether a contract address is equal to `address(0)` in an ether transfer function. When the address is zero, a new contract will be created instead of transferring ether. To avoid such cases, address validity checks should be performed. 10.8% dapp contracts and 7.4% template contracts contain transaction-reverting statements for address validity checks.

During our manual analysis, three clauses could not be categorized into the above sub-categories. Since they are not common, we do not further discuss them in the paper.

From the above analysis, we can see that dapp contracts and template contracts show differences in using transaction-reverting statements. 14.9% template contracts contain transaction-reverting statements for overflow/underflow checks, while the percentage in dapp contracts is only 3.9%. Besides, dapp contracts show higher percentages in using transaction-reverting statements for logic checks, range checks, and address validity checks. One possible reason is that developers will consider more specific factors when applying smart contracts to a real Ethereum environment, which could be complicated since a smart contract may need to interact with other contracts and user accounts. Comparatively, developers should consider general factors when developing template contracts and cannot anticipate the specific conditions that may arise in real environments.

Answer to RQ2: *Transaction-reverting statements are commonly used to perform authority verifications and validity checks, many of which involve security-critical constraints. Template contracts and dapp contracts have different purposes for using transaction-reverting statements.*

Implication: *Since transaction-reverting statements often check the runtime status of smart contracts against security-critical constraints, it is crucial to ensure the proper use of such statements. Future research can study the vulnerabilities induced by various misuses of transaction-reverting statements and propose detection or repairing techniques to combat such vulnerabilities.*

C. RQ3 (Developer Customization)

Study Methodology: As discussed earlier, many developers customize template contracts to develop their own smart contracts. In RQ3, we aim to understand how developers customize transaction-reverting statements in template contracts.

TABLE V
STATISTICS OF THE PURPOSES OF TRANSACTION-REVERTING STATEMENTS IN DAPP CONTRACTS AND TEMPLATE CONTRACTS

Category	First-Level Category	# Dapp Contracts	Ratio	# Template Contracts	Ratio
Authority Verification	Address Authority Check	65	14.9%	28	16.0%
	Token Verification	11	2.5%	3	1.7%
Validity Check	Logic Check	164	**37.7%**	51	**29.1%**
	Range Check	128	29.4%	47	26.9%
	Overflow/Underflow Check	17	**3.9%**	26	**14.9%**
	Arithmetic Check	3	0.7%	4	2.3%
	Address Validity Check	47	10.8%	13	7.4%
	Other	0	0.0%	3	1.7%
Total		435	100.0%	175	100.0%

TABLE VI
CONTRACT PRE-PROCESSING RESULT

	# Contracts	# Interfaces	# Libraries
Template contracts (270)	190	32	27
Dapp contracts (3,866)	4,563	1,191	484

Step 1: Mapping Template & Custom Contracts: To answer RQ3, the first step is to build a dataset containing template contracts and their corresponding custom contracts. However, real-world smart contracts rarely explicitly specify whether they are customized from a certain template or not. To address this problem, we leveraged code clone detection techniques to compute the similarities between each of our collected template contracts and the dapp contracts. We considered a dapp contract customized from a template contract if the two contracts have a high similarity.

A smart contract can contain multiple components, including contracts, libraries, and interfaces. In practice, different components are usually put in one file in dapp contracts, while in template contracts, a file usually contains a single component. To normalize the two kinds of contracts, we first broke down the dapp contracts into components and then compared the contracts at the component level. Table VI presents the result after this pre-processing step, where # Contracts, # Interfaces, and # Libraries represent the number of individual contracts, libraries, and interfaces, respectively.

We then adopted a code clone detector, SmartEmbed [20], to compare the dapp contracts with the template contracts. SmartEmbed computes similarities between two contracts based on word embeddings, and it has been shown to be effective in code clone detection for smart contracts. Following the original experiment setting of SmartEmbed, a dapp contract is considered a custom contract of a template contract if the similarity between these two contracts is higher than 85%. We chose this threshold because it achieves the highest recall in the evaluation of SmartEmbed.

As interfaces do not contain any statements, we excluded them from our dataset. In total, we obtained 175 contracts and 52 libraries from dapp contracts that are similar to template contracts. These contracts and libraries form the custom contracts dataset for our subsequent analysis.

Step 2: Detecting Customization Patterns: We leveraged the custom contracts to investigate how developers customized transaction-reverting statements. Inspired by an existing study that characterizes changes to `if` statements [24], we derived a taxonomy of possible customization patterns for transaction-reverting statements as shown in Table VII. Since transaction-reverting statements are also conditional statements, the change patterns of `if` conditional statements can also be applied to transaction-reverting statements. However, we found that patterns identified in the existing work (marked with * in Table VII) are not sufficient to cover all customizations on transaction-reverting statements. To identify more patterns, we manually analyzed 30% of the customized transaction-reverting statements. Specifically, we compared the transaction-reverting statements in the custom contracts with those in the corresponding template contracts and identified common customizations by checking the conditions of the transaction-reverting statements. Via the sampling and manual analysis, we identified five more patterns.

To investigate the prevalence of the customization patterns and identify commonly used ones, we implemented a static analyzer based on a Solidity parser [25] to automatically identify the occurrences of each customization pattern. For each pair of a template contract and a corresponding custom contract, we matched their functions by the function names and input parameters. Functions with the same name and input parameters are seen as matched function pairs. For each matched function pair, the analyzer 1) parses the source code into AST trees, 2) extracts all transaction-reverting statements, and 3) recognizes the customization patterns by comparing the syntactic differences of the transaction-reverting statements in the two functions. More details about the analyzer can be found on our project website [14].

Finding 4: *Transaction-reverting statements in template contracts are commonly customized. Developers are most likely to strengthen transaction-reverting statements by adding clauses, variables, or new transaction-reverting statements.*

Table VII shows the frequency of each customization pattern. In the 175 custom contracts and 53 custom libraries, our analyzer identified 529 occurrences of customization patterns. This indicates that developers commonly customize transaction-reverting statements in template contracts. 50.1% of the customizations are statement-level changes involving

TABLE VII
CUSTOMIZATION PATTERNS OF TRANSACTION-REVERTING STATEMENTS

Category	Customization Patterns	Description	Count	Percentage
Add	Add Clauses*	Add new clauses to a condition.	47	8.9%
	Add Variables*	Use new variables in the condition.	27	5.1%
	Add Statements	Add new transaction-reverting statements.	**161**	**30.4%**
	Sub-Total		**235**	**44.4%**
Delete	Delete Clauses*	Remove some clauses from a condition.	41	7.8%
	Delete Variables*	Remove some variables used in the condition.	24	4.5%
	Delete Statements	Delete some transaction-reverting statements.	**104**	**19.7%**
	Sub-Total		169	31.9%
Change	Modify Statement Types	Change transaction-reverting statements to other kinds of statements (e.g., change a `require` statement to an `if` statement) while the condition remains the same.	4	0.8%
	Modify Clauses	Make modifications to the clauses in the condition of a transaction-reverting statements.	59	11.2%
	Sub-Total		63	11.9%
Other	Cosmetic Changes	Modifications that do not change the semantics of the transaction-reverting statements.	62	11.7%
Total			529	100.0%

Note: Patterns marked with * are defined by an existing work [24]. The remaining patterns are newly identified by us.

the addition or removal of a transaction-reverting statement. 27.8% lie in clause granularity, including adding, deleting, and modifying a clause within the condition of a transaction-reverting statement. It is infrequent for developers to change transaction-reverting statements to other kinds of statements while keeping the condition unchanged (0.8%). In our dataset, all of the four cases in this category are changing transaction-reverting statements to general-purpose `if` statements.

44.4% of the customizations fall into the "add" category, 31.9% fall into the "delete" category, and 11.9% involved changes. Besides, 11.7% customizations are cosmetic changes, such as changing the order of clauses, adding or removing a string message, etc. These changes do not alter the semantics of the original transaction-reverting statements. These results show that developers are more likely to strengthen the transaction-reverting statements in template contracts.

Finding 5: *The customized transaction-reverting statements are commonly used for range checks and logic checks.*

We further investigated the purposes of customizing transaction-reverting statements. We randomly sampled 100 customized transaction-reverting statements and manually analyzed their purpose according to the taxonomy in Table IV.

Table VIII shows the analysis results. For the 100 statements, we identified 112 customized clauses and categorized them accordingly. The results indicate that the most frequent purposes of the customized transaction-reverting statements are *logic check* (30.4%), *range check* (48.2%), and *address validity check* (6.3%). This is reasonable since custom contracts may need to deal with use cases different from those encountered by template contracts. They would naturally have different definitions of the validity of runtime values. Figure 3 shows an example of a transaction-reverting statement in a custom contract.

```
require(
    _amount > uint256(0),
    "Stake amount must not be zero."
);
```

Fig. 3. An example customization with *Range Check* purpose

```
require(msg.sender == ceo, "CEO Only");
require(msg.sender == coo, "COO Only");
require(msg.sender == cfo, "CFO Only");
```

Fig. 4. An example customization with *Address Authority Check* purpose

It is a stake contract that allows EIP20 token to be staked. Staking is the process of investing tokens into the network and get a reward for doing it. Compared with the EIP20 token template contract, the custom contract adds a transaction-reverting statement to do *Range Check* to ensure that the staked amount provided by a staker is greater than 0, which intends to prevent the *Integer Underflow* vulnerability [26].

The other 17 (15.2%) customizations are related to *address authority check*, among which, ten added statements to perform authorization check on addresses and four deleted statements for address authority check. This is also understandable since custom contracts can have customized permission settings for different account types. For example, if there are multiple authorized users with different identities, the custom contract should add new transaction-reverting statements to verify the identity of the transaction sender to prevent unauthorized operations, as shown in Figure 4.

TABLE VIII
THE NUMBER OF USE PURPOSE FOR CUSTOMIZATION PATTERNS OF TRANSACTION-REVERTING STATEMENTS

Category	Sub-Category	# Total	Add			Delete			Modify		Other
			# Add Statements	# Add Clauses	# Add Variables	# Delete Statements	# Delete Clauses	# Delete Variables	# Modify Statement Types	# Modify Clauses	# Cosmetic Changes
Authority Verification	Address Authority Check	17	10	0	0	4	0	0	0	2	1
	Token Verification	0	0	0	0	0	0	0	0	0	0
Validity Check	Logic Check	**34**	**19**	0	0	**8**	0	0	0	1	6
	Range Check	**54**	**27**	1	4	**5**	3	2	0	6	6
	Overflow/ Underflow Check	0	0	0	0	0	0	0	0	0	0
	Arithmetic Check	0	0	0	0	0	0	0	0	0	0
	Address Validity Check	7	3	0	1	1	0	0	0	0	2
	Other	0	0	0	0	0	0	0	0	0	0
Total		112	59	1	5	18	3	2	0	9	15

Answer to RQ3: *Transaction-reverting statements in template contracts are commonly customized when developing smart contracts. Developers tend to strengthen transaction-reverting statements, mainly for logic and range checks.*

Implication: *The customizations of transaction-reverting statements often serve security purposes. Future research may also focus on investigating the security impact of the customizations of transaction-reverting statements.*

D. RQ4 (Security Impact)

Study Methodology: To answer RQ4, we mutated the dapp contracts by removing the transaction-reverting statements. We then leveraged smart contract security analyzers to detect the vulnerabilities in the original and the mutated contracts and compared the detection results. Specifically, we adopted a state-of-the-art framework, SmartBugs [27], to conduct the study. It integrates nine smart contract security analyzers, including HoneyBadger [28], Slither [29], Manticore [30], etc. Collectively, these nine analyzers can detect 141 types of vulnerabilities, while many of them refer to the same types of vulnerabilities but have different names. To unify the vulnerability types, we followed the existing practices [27] and used DASP [31], a smart contract vulnerability taxonomy, to categorize the reported vulnerabilities. Another problem with these analyzers is that they may generate many false alarms due to the imprecise static analyses [27]. To mitigate this problem, we followed the existing practice [27] and only counted those vulnerabilities that are reported by at least two of the nine analyzers.

Finding 6: *Missing transaction-reverting statements can introduce security vulnerabilities to smart contracts.*

Table IX presents the number of original dapp contracts and mutated contracts that are reported to contain vulnerabilities. The number of vulnerable contracts increases after removing the transaction-reverting statements. In particular, the number of contracts containing *Time Manipulation* and *Front Running* vulnerabilities increase significantly, by 16.98% and 12.90%, respectively. This shows that transaction-reverting statements are useful in improving the security of smart contracts. To ease understanding, we provide an example. Figure 5 shows a code snippet in a real smart contract. After

TABLE IX
THE NUMEBER OF CONTRACTS WITH AT LEAST ONE VULNERABILITY DETECTED BY MULTIPLE ANALYZERS

Vulnerability Category	# Before	# After	Increase Ratio
Access Control	3,857	3,857	0.00%
Arithmetic	1,871	2,041	9.09%
Denial Service	174	178	2.30%
Reentrancy	634	671	5.84%
Unchecked Low Calls	254	255	0.39%
Front Running	550	621	**12.90%**
Time Manipulation	132	159	**16.98%**
Unknown Unknowns	690	738	6.96%

"# Before" and "# After" present the security vulnerability detection results for the original contracts and the mutated contracts, respectively.

```
1 function requireCorrectReceipt(uint offset) view
      private {
2     uint leafHeaderByte; assembly { leafHeaderByte
          := byte(0, calldataload(offset)) }
3     require (leafHeaderByte >= 0xf7, "Receipt leaf
          longer than 55 bytes.");
4     offset += leafHeaderByte - 0xf6;
5     ...
6 }
```

Fig. 5. The number of vulnerabilities increases after mutation in contract 0x9F91b5Aa41b9fbDae6877593910586484d291F05.

removing the transaction-reverting statement in Line 3, the contract is reported to have an *Underflow/Overflow* vulnerability [26]. In this example, both `leafHeaderByte` and `offset` are unsigned integers. If Line 3 is removed, the value of `leafHeaderByte` in Line 4 can be smaller than `0xf7`, which may lead to an underflow.

Finding 7: *Smart contract security analyzers can fail to analyze transaction-reverting statements properly and induce false negatives in security vulnerability detection.*

When inspecting the results reported by the nine analyzers, we found that there are also cases where vulnerabilities in original smart contracts disappear after removing transaction-reverting statements. This is counter-intuitive as we have found that transaction-reverting statements are commonly used for security checks. We found out that eight out of the nine contract analyzers (except Maian [32]) used in our study suffered from this problem. We further inspected such cases

```
1 function contributeWithAddress(address contributor)
      payable {
2     require(msg.value >= minContribAmount);
3
4     uint contribValue = msg.value;
5     uint oldTotalContributed = totalContributed;
6     totalContributed = oldTotalContributed.add(
          contribValue);
7     uint newTotalContributed = totalContributed;
8
9     if (newTotalContributed >= softCapAmount &&
10        oldTotalContributed < softCapAmount)
11    {
12        softCapReached = true;
13        endTime = afterSoftCapDuration.add(now);
14        onSoftCapReached(endTime);
15    }
16    ...
17 }
```

Fig. 6. The number of vulnerabilities decreases after mutation in contract 0x0AbdAce70D3790235af448C88547603b945604ea.

identified in our dataset.

Figure 6 shows a code snippet in a real smart contract. The function `contributeWithAddress()` is reported to have a *Timestamp Dependence* vulnerability [33]. Due to the direct use of `now` (Line 16) which is an alias of `block.timestamp`, a malicious block miner can manipulate the block's timestamp to gain profits from the contract. In this case, the transaction-reverting statement in Line 2 is not related to the vulnerability as it is not checking against block timestamp. In other words, after removing it, the *Timestamp Dependence* vulnerability should still exist. However, the tool Osiris [34] does not report the vulnerability after removing this transaction-reverting statement. This shows that Osiris can be fooled by the removal of transaction-reverting statements and induce false negatives.

We observed 5,404 such cases in our dataset where the originally detected vulnerabilities disappeared after removing transaction-reverting statements. In our future work, we plan to take a deeper look into this problem and investigate why removing transaction-reverting statements can fool smart contract security analyzers.

Answer to RQ4: *Missing transaction-reverting statements can induce security vulnerabilities in smart contracts. In other words, transaction-reverting statements can be used to avoid vulnerabilities effectively. However, there are also cases where removing irrelevant transaction-reverting statements can fool smart contract analyzers and induce false negatives in security vulnerability detection.*

Implication: *Researchers need to further improve the effectiveness of smart contract security analyzers. In particular, properly dealing with transaction-reverting statements is a basic and critical requirement for such tools.*

V. Threats to Validity

The validity of our study results may be subject to several threats. First, our selected template contracts may not be sufficiently diverse or representative. To mitigate this threat, we considered the popularity of the templates in the selection process. The four template contract repositories are all widely used by developers on GitHub [35]. Second, we proposed a taxonomy to categorize the purposes of using transaction-reverting statements in Table IV. There could be other ways to categorize the purposes. To address this threat, we followed the widely-used open coding procedure to derive the results. Third, our study results may be affected by human subjectivity, which is a common problem in qualitative coding [36]. To reduce this threat, we followed the common research practices on manual labeling by involving multiple people. Three authors iterated the labeling process three times to obtain the final taxonomy. This helped improve the reliability and generality of our taxonomy. Our data is also released for public scrutiny [14]. Fourth, we used a code clone technique, SmartEmbed [37], to identify custom contracts of template contracts and set the code similarity threshold as 85% following the experiments in the original paper to reduce false negatives. The chosen code clone technique and threshold may affect the mapping results. Also, the subjects used when investigating RQ3 are limited. We will keep expanding our dataset in the future and trying other clone detectors to see if more reliable results can be obtained. Lastly, we used a framework supporting nine smart contract security analyzers to detect vulnerabilities in RQ4. False positives and false negatives can both exist in the results. To reduce the threat, we only kept results for items detected as vulnerable by more than one analyzer. Besides, the quality of transaction-reverting statements used in our constructed contract dataset may affect the accuracy of the results since our analysis is based on the assumption that the transaction-reverting statements analyzed are correct. We plan to conduct more experiments and analyses in future studies to validate our findings further.

VI. Related Work

Error-handling Statements. Various studies have been conducted to characterize error-handling statements in other areas. Filho et al. [38] studied the impacts of factors that affect the exception handling code in aspect-oriented programming (AOP) techniques. Tian et al. [39] conducted a comprehensive study of error-handling bugs and their fixes and implemented *ErrDoc*, a tool to diagnose and repair error-handling bugs in C programs automatically. Some other studies [40]–[44] automatically detected and patched error-handling bugs using a variety of techniques. Different from the previous studies, our work conducts the first empirical study on transaction-reverting statements (a type of error-handling statements) for Ethereum smart contracts. It reveals the security impact of transaction-reverting statements, which is specific to smart contracts.

In terms of smart contracts, several previous studies have discussed the usefulness of transaction-reverting statements

for providing defenses for vulnerabilities. Xue et al. [45] showed that the `require` statement could be used to prevent reentrancy vulnerability. Zhou et al. [46] observed that most smart contracts implemented defenses via transaction-reverting statements to abort a transaction when noticing an attack. However, these studies only reported the use cases of transaction-reverting statements for specific purposes and did not regard them as their major focuses. In comparison, our work is the first empirical study that systematically characterizes the use of transaction-reverting statements in real-world smart contracts.

Smart Contract Vulnerability Detection. In recent years, there have been many studies targeting smart contract vulnerability detection. Static analysis methods inspected the code of smart contracts without executing them. Examples are *Oyente* [47], *Zeus* [48], *Vandal* [49], *Securify* [50], *F* Framework* [51], and *Fether* [52]. Dynamic analysis methods check the runtime behavior of smart contracts to detect vulnerabilities. Nikolic et al. [53] employed inter-procedural symbolic analysis and concrete validators for detecting real security vulnerabilities. Ting et al. [54] constructed three kinds of graphs to characterize major activities on Ethereum and proposed graph-based techniques to detect security issues. While these studies proposed different techniques to detect vulnerabilities in smart contracts, none discussed the security impact of transaction-reverting statements. Our work showed the prevalence of transaction-reverting statements and concluded the security impact of such statements. Our findings can help improve security vulnerability detection techniques for smart contracts.

VII. Conclusion and Future Work

In this work, we present the first empirical study on transaction-reverting statements in Ethereum smart contracts. Through intensive analyses of 3,866 real-world smart contracts and 270 popular template contracts, we showed that transaction-reverting statements are prevalent in smart contracts. They are often used to check the runtime status of smart contracts against security-critical constraints. Our study characterizes the usage of transaction-reverting statements in practice and may shed light on future research in areas such as smart contract security and quality assurance.

In the future, we plan to extend our study by investigating the challenges in properly using transaction-reverting statements and identifying security issues induced by the misuse of transaction-reverting statements. We also plan to leverage our findings to improve the security vulnerability detection techniques for smart contracts.

Acknowledgment

This work was supported by the National Natural Science Foundation of China (Grant No. 61932021 and No. 62002125), Hong Kong RGC/GRF (Grant No. 16207120), Hong Kong RGC/RIF (Grant No. R5034-18) and Guangdong Provincial Key Laboratory (Grant No. 2020B121201001). Lili Wei was supported by the Postdoctoral Fellowship Scheme of the Hong Kong Research Grant Council.

References

[1] "Top 12 smart contract use cases," https://101blockchains.com/smart-contract-use-cases/.
[2] "Ethereum: A secure decentralized generalized transaction ledger," https://ethereum.github.io/yellowpaper/paper.pdf.
[3] "Coinmarketcap. top 100 cryptocurrencies by market capitalization," https://coinmarketcap.com/.
[4] "Ethereum daily transactions chart," https://etherscan.io/chart/tx.
[5] "Solidity documentation v0.8.3," https://docs.soliditylang.org/en/v0.8.3/.
[6] "Stack overflow website," https://stackoverflow.com/.
[7] "Swc registry," https://swcregistry.io/.
[8] "Swc 104: Unchecked call return value," https://swcregistry.io/docs/SWC-104.
[9] "Openzeppelin contract library," https://github.com/OpenZeppelin/openzeppelin-contracts.
[10] "Aragonos smart contract framework," https://github.com/aragon/aragonOS.
[11] "Ethereum smart contract best practices," https://github.com/ConsenSys/smart-contract-best-practices/.
[12] "Ethereum improvement proposals," https://eips.ethereum.org/erc.
[13] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.
[14] "Dataset for 'characterizing transaction-reverting statements in ethereum smart contracts'," https://github.com/transaction-reverting-statements/Characterizing-require-statement-in-Ethereum-Smart-Contract.git.
[15] 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.
[16] "Ethereum in bigquery: a public dataset for smart contract analytics," https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics.
[17] "Dapp.com," https://www.dapp.com/.
[18] "Uniswap," https://uniswap.org/.
[19] "Etherscan," https://etherscan.io/.
[20] Z. Gao, L. Jiang, X. Xia, D. Lo, and J. Grundy, "Checking smart contracts with structural code embedding," *IEEE Transactions on Software Engineering*, 2020.
[21] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *Proceedings of the 2012 International Conference on Software Engineering*, 2012, pp. 102–112.
[22] J. Harty, H. Zhang, L. Wei, L. Pascarella, M. Aniche, and W. Shang, "Logging practices with mobile analytics: An empirical study on firebase," *arXiv preprint arXiv:2104.02513*, 2021.
[23] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.
[24] K. Pan, S. Kim, and E. J. Whitehead, "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.
[25] "Python solidity parser," https://github.com/ConsenSys/python-solidity-parser.
[26] "Swc 101: Integer overflow and underflow," https://swcregistry.io/docs/SWC-101.
[27] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the 2020 ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 530–541.
[28] C. F. Torres, M. Steichen *et al.*, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *Proceedings of the 2019 28th Usenix Security Symposium*, 2019, pp. 1591–1607.
[29] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *Proceedings of the 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2019, pp. 8–15.
[30] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *Proceedings of the 2019 IEEE/ACM 34th International Conference on Automated Software Engineering*, 2019, pp. 1186–1189.
[31] "Decentralized application security project(dasp)," https://dasp.co/.
[32] "Maian," https://github.com/ivicanikolicsg/MAIAN.

[33] "Swc 116: Block values as a proxy for time," https://swcregistry.io/docs/SWC-116.

[34] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 664–676.

[35] "Github," https://github.com/.

[36] Y. Chandra and L. Shang, *Qualitative research using R: A systematic approach*. Springer, 2019.

[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 *Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution*, 2019, pp. 394–397.

[38] F. Castor Filho, A. Garcia, and C. M. F. Rubira, "Extracting error handling to aspects: A cookbook," in *Proceedings of the 2007 IEEE International Conference on Software Maintenance*, 2007, pp. 134–143.

[39] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in c," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 752–762.

[40] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes," in *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, 2004, pp. 419–431.

[41] M. Susskraut and C. Fetzer, "Automatically finding and patching bad error handling," in *Proceedings of the 2006 Sixth European Dependable Computing Conference*, 2006, pp. 13–22.

[42] J. Lawall, B. Laurie, R. R. Hansen, N. Palix, and G. Muller, "Finding error handling bugs in openssl using coccinelle," in *Proceedings of the 2010 European Dependable Computing Conference*, 2010, pp. 191–196.

[43] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *Proceedings of the 2016 25th USENIX Security Symposium*, 2016, pp. 345–362.

[44] Z. Jia, S. Li, T. Yu, X. Liao, J. Wang, X. Liu, and Y. Liu, "Detecting error-handling bugs without error specification input," in *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 213–225.

[45] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 1029–1040.

[46] S. Zhou, M. Möser, Z. Yang, B. Adida, T. Holz, J. Xiang, S. Goldfeder, Y. Cao, M. Plattner, X. Qin *et al.*, "An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem," in *Proceedings of the 2020 29th Usenix Security Symposium*, 2020, pp. 2793–2810.

[47] 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.

[48] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *Proceedings of the 2018 ISOC Network and Distributed System Security Symposium*, 2018, pp. 1–12.

[49] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[50] 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.

[51] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy *et al.*, "Formal verification of smart contracts: Short paper," in *Proceedings of the 2016 ACM workshop on programming languages and analysis for security*, 2016, pp. 91–96.

[52] Z. Yang and H. Lei, "Fether: An extensible definitional interpreter for smart-contract verifications in coq," *IEEE Access*, vol. 7, pp. 37 770–37 791, 2019.

[53] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 2018 34th Annual Computer Security Applications Conference*, 2018, pp. 653–663.

[54] 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*, vol. 20, no. 2, pp. 1–32, 2020.