When They Go Low: Automated Replacement of Low-level Functions in Ethereum Smart Contracts

Rui Xi
Department of Electrical and Computer Engineering
University of British Columbia (UBC), Vancouver, Canada
xirui801@student.ubc.ca

Karthik Pattabiraman
Department of Electrical and Computer Engineering
University of British Columbia (UBC), Vancouver, Canada
karthikp@ece.ubc.ca

***Abstract*—Smart contracts in the Ethereum blockchain are typically written using a high-level, Turing-complete language called Solidity. However, the Solidity language has many features to allow programmers fine-grained control over their smart contracts. We call these features *low-level functions*. Unfortunately, the improper use of low-level functions can lead to security vulnerabilities leading to heavy financial losses. Therefore, the Solidity community has suggested alternatives for the low-level functions in the official guidelines for developers.**

We first perform a large-scale empirical study on the use of low-level functions in Ethereum smart contracts written in Solidity. We find that such functions are widely used in real-world Ethereum smart contracts, and that the majority of these uses are gratuitous for the smart contract's functionality. We then propose GoHigh, a source-to-source transformation tool to eliminate low-level function-related vulnerabilities, by replacing low-level functions with high-level alternatives. We evaluate GoHigh on over $300,000$ real-world smart contracts on the Ethereum blockchain. GoHigh replaces all low-level functions that are amenable to replacement in the contracts with 17% fewer compiler warnings, and the externally-visible behaviors of at least 92% of the replaced contracts are identical to the original ones. Finally, GoHigh takes 7 seconds on average per contract.

***Index Terms*—Ethereum blockchain, Smart contract, Source-to-source transformation, Static analysis**

I. Introduction

The Ethereum blockchain has grown to over 250 billion U.S. dollars in April 2021. One of the defining features of Ethereum are smart contracts, which are small programs that can be executed on the Ethereum blockchain to perform transactions. There are more than 1.5 million [1] smart contracts on Ethereum, and some of these have attracted millions of transactions. Smart contracts are usually written using Solidity, a Turing-complete language, and compiled down to an executable format known as Ethereum Virtual Machine (EVM) bytecode before deploying them on the blockchain.

Unfortunately, smart contracts are attractive targets for attackers, as they deal directly with transacting money (i.e., Ether, which is a virtual crypto-currency in Ethereum). Further, because smart contracts can be called by anyone who has the address of the contract, and the resulting transfer of Ether is automated, there are few restrictions on the attackers. These factors have led to many attacks on smart contracts [2], [3], which have resulted in losses of millions of dollars.

The increasing security attacks on smart contracts have prompted the Solidity community to identify insecure code patterns that are liable to lead to security vulnerabilities [4], and to warn developers against using them [5]. However, for backward compatibility, the old insecure patterns are still supported. Further, once a smart contract is deployed, it is very difficult to update or modify it, and hence the old, insecure patterns continue to proliferate. This is exacerbated by high levels of code reuse in smart contracts [6], [7]. Finally, Solidity developers often do not follow the advise to avoid the patterns, either due to ignorance or inertia.

In this paper, we characterize the use of insecure patterns that we term *low-level functions* in Solidity. We first perform a detailed analysis of the insecure code patterns that are mentioned in the Solidity documentation, and that have secure replacements (i.e., high-level alternatives). However, we find that these low-level functions are widely used in real-world smart contracts on the Ethereum blockchain. We further find that the majority of the uses of the low-level functions (about 80%) are gratuitous for the smart contract's functionality, and can hence be replaced by high-level alternatives. However, there are many different patterns of such low-level functions, which make them challenging to replace in a uniform manner.

Based on the above insight, we then build an automated static replacement engine, GoHigh [1] that works on the abstract syntax tree (AST) of the smart contract's source code to replace low-level functions with high-level alternatives. We distill the different patterns of low-level functions that are commonly used in smart contracts into AST templates, and propose replacements for the patterns taking into account their unique constraints, without requiring any programmer effort.

GoHigh covers multiple classes of security vulnerabilities in smart contracts detected by prior work [4], [8], [9]. None of these papers propose any solution to deal with the vulnerabilities, and leave it up to the developer to rewrite the contracts. In comparison, GoHigh addresses both the unhandled exception-related vulnerabilities [4], [8], [9] and the use of low-level and obsolete functions vulnerabilities [9], without requiring any effort from the developer. SmartShield [6] and EVMPatch [10] leverage bytecode rewriting to protect vulnerable fields in a

[1]The name GoHigh as well as the paper's title are inspired by a quote from Michelle Obama in 2016, "When they go low, we go high".

smart contract. However, these techniques introduce additional checks at runtime, which can incur high performance overheads. Further, they require either manual or semi-supervised source code analysis methods to identify vulnerabilities before performing bytecode level rewriting, which require programmer effort. Finally, the checks added by these techniques mostly focus on who can access the smart contract (*i.e.*, access control), rather than the root causes of the vulnerabilities.

To the best of our knowledge, we are the first paper to empirically characterize the use of low-level functions in Ethereum smart contracts, and propose an automated technique to statically replace them with high-level alternatives. In summary, we make the following contributions in this paper.

- We define a low-level function based on the Solidity documentation, and empirically analyze the usage of low-level functions in a real-world dataset consisting of nearly $150,000$ Ethereum smart contracts ("base dataset"). We find that about 14% of the contracts in the base dataset use low-level functions, and that about 80% of these uses are gratuitous for the smart contract's functionality.
- We mine the base dataset to distill 11 distinct patterns of low-level functions that are used in the contracts, and represent the patterns as AST structures.
- We develop an automated, source-to-source transformation tool called GoHigh to detect low-level functions corresponding to the AST structures, and to replace the low-level functions with high-level alternatives.
- We also analyze a more recent dataset of Ethereum smart contracts consisting of over $150,000$ contracts ("recent dataset"). We find that the number of smart contracts using low-level functions more than doubled (37%) compared to the base dataset, and more than 75% of these uses were gratuitous for the contract's functionality.
- We find that the patterns distilled from the base dataset also applied to *all the contracts* in the recent dataset. Further, GoHigh reduces the number of compiler warnings in the contracts by 17% and 38% respectively for the base and recent datasets. We also compare the behavior of the contracts after replacement by GoHigh by replaying Ethereum transactions on the original and replaced contracts, and checking whether their externally visible states match. We find that 95% and 92% of the state changes matched after replacement by GoHigh in the base and recent datasets respectively (we could not validate the remaining contracts due to experimental limitations). Finally, GoHigh takes 7 seconds per contract on average.

II. Low-Level Functions: Definition and Analysis

In this section, we first define *low-level functions*, and present the criteria of our selection of the same by distilling the Solidity documentation. We then delve into the three categories of low-level functions considered in this work.

A. Selection and Definition of Low-level Functions

We define a low-level function as one that has a warning against its use in the Solidity documentation, and can be detected and replaced by syntax-level analysis. To extract low-level functions, we systematically study the official Solidity documentation[1] and collect all the alerts highlighted in warning boxes in the *Language Description* section of the documentation (the other sections have to do with tutorials for beginners and implementation details of the EVM, neither of which are relevant to our study).

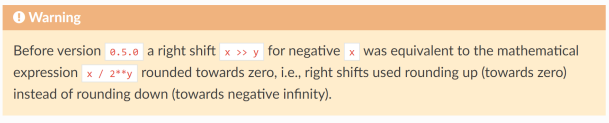

Fig. 1: Example of a warning box in Solidity documentation.

We find that there are 28 warning boxes in total in the documentation[2]. Of the 28 warnings, we observe that seven warnings (W2-3, W7, W16, W19, W24-25) are constrained to a specific range of compiler versions We exclude such warnings as they can be eliminated with later versions of Solidity compilers. For example, in Figure 1, the **W2** says "Before version 0.5.0, a right shift ...", which means that it is confined to Solidity compilers version 0.4.x or earlier.

We then group the remaining 21 warnings into seven categories, *i.e.*, Arithmetic Operations (AO), Deprecated `Send` (DS), Low-level `call` (LC), Inline Assembly (IA), Cryptographic Functions (CF), Block Timestamp (BT) and Contract States (CS). Table I shows these categories. These categories are based on the corresponding operations. For example, warnings about the send function fall into the DS category, and warnings about arithmetic operations (*e.g.* , add, subtract, shift and etc.) fall into the AO category.The LC category includes warnings about low-level call family, while the IA category contains all warnings from the Inline Assembly subsection. The CF category and BT category are about the misuse of the *block timestamp* attribute and the *ecrecover* cryptographic function, respectively. The CS category includes operations that influence the contract states, such as the contract's balance, state visibility, and self-destruct function.

We follow three criteria to select the low-level functions to focus on in this work. First, there should not be widely available, mature solutions to the category, *e.g.* , a practical and widely-used library. Second, the category should have real-world exploits. Some vulnerabilities are in functions that are seldom used by developers, and thus have no real-world exploits. Third, the problem must be addressable with simple syntactic replacement, and not require semantic knowledge of the contract. This is because, as a static analysis technique, our technique has no information about the contract's semantics.

Based on the above criteria, we choose three categories of warnings as low-level functions, namely DS, LC and IA. DS and LC both operate on addresses in Solidity, and share many similarities in their vulnerabilities, *i.e.*, mishandled exception and existence check. The documentation also suggests alternatives (e.g., transfer for send, high-level calls that operate on contract instances for low-level calls) to replace them.

[1] https://docs.soliditylang.org/en/v0.8.6

[2] We assign numbers to the warnings in the order of their appearance.

TABLE I: Different Categories of Warnings in the Solidity documentation, and their characteristics for inclusion in our study.

Name	Included	Vulnerabilities	Warnings	Criteria for Inclusion		
				No Mature Solutions	Real-world Exploits	No Semantic Knowledge
AO	✗	Underflow/overflow	W1, W4, W17	✗	✓	✓
DS	✓	Mishandled exception Existent check Out-of-gas	W5, W10, W21	✓	✓	✓
LC	✓	Mishandled exception Reentrancy Existent check Argument Invalid	W6, W11-13, W18	✓	✓	✓
IA	✓	Same as LC's	W26-28	✓	✓	✓
CF	✗	Not unique signature	W8	✓	✗	✓
BT	✗	Bad randomness	W14	✓	✓	✗
CS	✗	Balance mismatch Hidden state exposure Unexpected copy behavior	W15, W20, W22-23	✓	✓	✗

There have also been real-world vulnerabilities in both of these categories. For example, the DAO attack[3] exploited the reentrancy vulnerability in a low-level call and resulted in a hard-fork in Ethereum. Therefore, we include the DS and LC categories. We also include the IA category, as there is also a call instruction in inline assembly which is similar to the LC category, and thus it shares the same vulnerabilities as LC. Although there are no real-world exploits in the IA category, many attacks in the DS and LC categories also apply to it.

We briefly describe why we exclude the other categories. AO has been extensively studied in the research community [11], [12], and can be easily mitigated by using the arithmetic functions in the Solidity math library named SafeMath [13]. There has also been extensive work on byte-code level rewriting [14], [15] to use such techniques. BT requires understandings of a contract's semantics for its mitigation. A block timestamp is a number filled by the block miner, and can be modified during block generation, which makes it vulnerable to malicious miners. While it is recommended to avoid using BT as a source of randomness, the specifics of when such use is acceptable depends on the contract's semantics. Similarly, CS require semantic knowledge for mitigation to determine whether a state of the contract is hidden. Therefore, we exclude these categories. Finally, we exclude CF as we could not find any real-world exploits in this category.

In the following subsection, we describe the three categories of warnings in our definition of low-level functions in this paper: DS, LC and IA. *In the rest of this paper, we refer to the warnings to avoid their use as Solidity guidelines.*

B. Low-Level Functions Included

1) Deprecated `send` (DS): Deprecated `send` is a member function of an address object that is used to send ether to an address. Address is a built-in data type of Solidity, which supports a series of functions that directly interact with this address. By invoking DS, the contract sends Ether to the targeted Ethereum account (`address receiver` in Fig.2a). However, a failed transaction via DS does not trigger a rollback. Instead, it returns `false`, so the execution continues even though the transaction fails. As a result, from the Solidity version 0.4.10 onwards, the documentation recommends that developers use a safer replacement for send (e.g. `transfer`).

2) Low-level `call` (LC): Low-level `call` is a member function of an address to call a function in this address (if any). It is used to call an arbitrary function, similar to a function pointer in C and C++. It is primarily used to interface with contracts that do not adhere to the Application Binary Interface (ABI), or to get more direct control over the encoding. There are three functions for performing such LCs, namely `call`, `staticcall` and `delegatecall`. Figure 2b shows an example of an LC: it takes a single byte memory parameter (payload in the example) and returns the success condition (as a `bool`) as well as the data (`bytes memory`). We focus on the `call` function, as it is used by 98.7% of contracts using the call functions in our smart contract dataset (Section III).

3) Inline Assembly (IA): Solidity allows developers to have fine-grained control on the memory allocation in contracts via IA, which is close to the EVM. Developers leverage this feature to save gas cost, especially for frequently-used contracts. IA also provides a `call` instruction. Fig.2c shows an example of a call in IA - it calls a contract using a hand-crafted payload (`calldata`) at a given address (`childContract`), returning 0 on error (*e.g.* , out of gas) and 1 on success[4]. Crafting the payload may result in corrupted data if the encoding is not correctly implemented. Moreover, the call instruction can also be used to invoke arbitrary function calls by the smart contract, which lead to the same issues as LCs.

III. Empirical Study

We perform an empirical study of smart contracts deployed on the Ethereum blockchain to understand the frequency and the usage of low-level functions in real-world smart contracts. We first describe the dataset and tools, followed by the results.

A. Dataset

We use a real-world smart contracts published to the Ethereum blockchain, from March 2016 to September 2019[5] Of the 16 million contracts in this dataset, 149,363 contracts have their source code available - these constitute our dataset.

[3]https://www.coindesk.com/understanding-dao-hack-journalists

[4]https://docs.soliditylang.org/en/v0.8.6/yul.html#yul-call-return-area

[5]We use a more recent dataset to evaluate GoHigh in Section V.

```
address payable receiver = address(0x123);
receiver.send(10);
```

(a) An example of deprecated send.

```
address nameRegister = address(0x123);
bytes memory payload = abi.encodeWithSignature("
    register(string)", "John");
nameRegister.call(payload);
```

(b) An example of low-level call.

```
//0x095ea7b3 == "approve(address,uint256)"
bytes memory calldata = abi.encodeWithSelector(0
    x095ea7b3, this, _childTokenId);
assembly {
    let success := call(gas, childContract, 0, add(
        calldata, 0x20), mload(calldata), calldata
        , 0)
}
```

(c) An example of inline assembly call.

Fig. 2: Examples of low-level functions.

All of the contracts specify Solidity compiler versions from 0.4.0 to 0.5.4. The version information for a contract is important, as the contract will not compile if the compiler version does not match the declared one (in some cases, no compiler version is specified, and so we try compiling it with different versions to see if there is a match). Note that the guidelines in the documentation were introduced for Solidity compiler 0.4.0 and evolved from 2016 to 2018, before being finalized in November 2018 with the release of Solidity 0.5.0. Thus, these contracts were developed more or less simultaneously with the guidelines in the documentation.

We determine whether a contract is unique based on its MD5 checksum, as it has been shown that many contracts in Ethereum are duplicated [6], [7]. We find that there are $61,444$ unique contracts in our dataset (41% of the dataset). Therefore, we present results for both the unique and overall contracts.

B. Tools

We quantify the usage of low-level functions by collecting smart contract addresses using Ethereum-ETL[6] and performing offline analysis subsequently. The source code of the contracts is collected from Etherscan[7], a public platform that monitors every transaction recorded in Ethereum, including the creation of smart contracts and users' interaction with them.

C. Results

We start by studying the frequency of low-level functions in the smart contracts in the dataset in Fig.3a. *We find that among* $149,363$ *contracts,* 13.8% *use low-level functions.* The percentage increases slightly to 17.2% if we consider the set of unique contracts, as shown in Fig.3b. These figures show that low-level functions are widely used in smart contracts.

We further perform detailed analysis of low-level functions. Before doing so, we subdivide low-level `calls` (LCs) into two sub-categories: (1) constant string LC and (2) arbitrary LC. The former refers to cases where the developer hard-codes the function to be called into a constant string, and invokes the function using an LC. The latter refers to cases where the call is made to an arbitrary function whose address is difficult to determine at compile-time. Figures 4a and b show examples of the constant string LC and the arbitrary LC, respectively.

[6]https://github.com/blockchain-etl/ethereum-etl

[7]https://Etherscan.io/

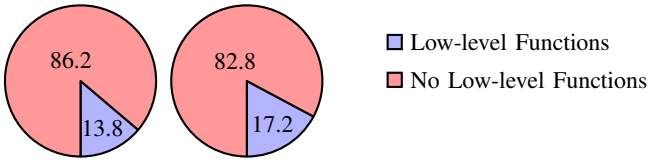

Fig. 3: Low-level functions usage in the base dataset in (a) all contracts, and (b) unique contracts

```
function constant_string_call() public{
    address hardcode_address = address(0x123);
    bytes memory hardcode_payload = abi.
        encodeWithSignature(
        "register(string)", "John");
    hardcode_address.call(hardcode_payload);
}
```

(a) An example of constant string LC.

```
function arbitrary_call(address arbitrary_address,
    bytes memory arbitrary_payload) public{
    arbitrary_address.call(arbitrary_payload);
}
```

(b) An example of arbitrary LC.

Fig. 4: Examples of two subcategories of LCs.

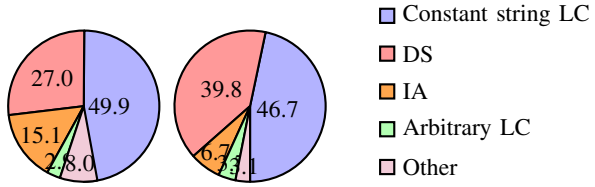

Fig. 5: The distribution of low-level functions in the base dataset in (a) all contracts, and (b) unique contracts.

We calculate a more detailed usage statistics of low-level functions in Fig.5, including the two sub-categories of LCs. We only consider the contracts that use at least one low-level function in this figure. As shown in Fig.5a, the constant string LC is used by the dominant fraction (49.9%) of all the contracts that use low-level functions. The DS is in second place at 27.0% and the IA is at 15.1%. "Other" in this figure stands for the contracts that include more than one low-level function categories. Note that the arbitrary LC is only found in 2.9% of the dataset. If projected to the entire dataset of smart contracts on Ethereum, the percentage of contracts that use arbitrary LC is only 0.40% ($2.9\% \times 13.8\%$).

Figure 5b shows the detailed breakdown of low-level function usage among the unique contracts. We find that the percentage of DS increases from 27.0% to 39.8% in this dataset, while the percentage of IA halves, representing 6.7%. The percentage of arbitrary LC slightly increases from 2.9% to 3.7%, but it is still the least prevalent among the four categories, as was the case with the overall contracts.

D. Summary

We can make two key observations from the results.

1) Low-level functions are widely used in real contracts.
2) The majority of low-level functions are gratuitous and can be replaced by high-level ones.

The first observation follows directly from the results on the prevalence of low-level functions in both the overall contract set as well as the set of unique contracts on Ethereum that have their source code available. The second observation follows from the distribution of the low-level function usage in the dataset. For example, most of the low-level functions are in the constant string LC category, and can hence be replaced using high-level call functions that directly call the function encoded in the constant string. Similarly, the DS calls can be replaced by the `transfer` function. Together, these two categories constitute 76.9% of the set of all contracts, and 86.5% of the set of unique contracts on Ethereum. If we include the Other category, the percentage of contracts using low-level functions that can be statically replaced increases to 82%.

Therefore, only 18.0% *of smart contracts using low-level functions cannot be statically replaced with high-level alternatives (*15.1% *IAs plus* 2.9% *arbitrary LCs).* With that said, not all contracts in the remaining 82% are straight-forward to replace, as we describe in the next section.

IV. METHODOLOGY

In this section, we present GoHigh, an automated technique to transform the source code of Ethereum smart contracts, and replace low-level functions with their high-level alternatives. GoHigh has three steps. First, it converts the Solidity source codes of smart contract into AST. Then, it searches the AST for 11 patterns of DS and constant string LC, which we extracted from the dataset of contracts containing low-level functions. Finally, GoHigh replaces the matching sub-tree in the AST with high-level alternatives.

We start by explaining the challenges in replacing the low-level functions by GoHigh. We then explain each of the above steps, and finally provide an example of how GoHigh works.

A. Challenge and Contribution

Challenge:. Developers tend to use home-grown checks to prevent the vulnerabilities of low-level functions. For instance, developers may require the return value of an LC to be true, and revert the whole transaction if the call fails. However, these checks may be incomplete or seem right but shadow other vulnerabilities. One general issue is that the patterns do not check the existence of the callee function. If callee functions do not exist, the return value of the LC is always True, which bypasses all the home-grown checks. Moreover, hard-encoded strings may also be incorrect (e.g., typos, missing parameters).

The main challenge is that one replacement does not work on all patterns of use of the low-level functions. Fig.6 shows an example, where a direct replacement works for the first pattern but does not work for the second pattern. For the first pattern (Fig.6a), the replacement strategy is to extract the function name and the parameters and reassemble them into a high-level call. However, if the same replacement strategy is applied to the second pattern (Fig.6b), the expression in the if-clause is replaced with a high-level call. Unfortunately, the new if-statement may not pass compilation because the return value of the high-level call may not be a Boolean value (or there may be no return value), and thus cannot serve as the operand of the logical negation operator. Therefore, we need to come up with custom replacement strategies for each pattern.

```
address(0x123).call(abi.encodeWithSignature(
    "register(string)", "John")); //before
address(0x123).register("John"); //after
```

(a) A simple pattern and its replacement.

```
if(!address(0x123).call(abi.encodeWithSignature(
    "register(string)", "John"))){
    revert();} //before
if(!address(0x123).register("John")){ //fail
    compilation
    revert();} //after
```

(b) Applying the direct replacement to a complex pattern.

Fig. 6: One replacement does not apply to every pattern.

Our Contribution: We distill the patterns of low-level functions manually from our dataset, so that GoHigh can automatically identify the patterns at the AST level. The 11 patterns are combinations of three representations (shown in Table II) and five patterns (shown in Table III). Three of the five patterns exist in all three representations, resulting in nine combinations; two of five patterns exist only in the send representation, resulting in the remaining two combinations.

B. AST Generation

To extract the patterns, GoHigh first converts the source code into an AST representation using the Solidity compiler, `solc`. An example AST is shown in Fig.7. The root node of the tree is a SourceUnit, and its nodes correspond to different syntactic elements in the smart contract.

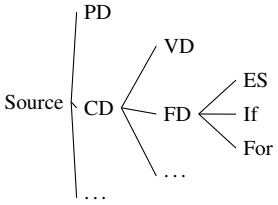

Fig. 7: The structure of an AST of a Solidity smart contract. PD stands for PragmaDirective, CD stands for ContractDefinition, VD stands for VariableDeclaration, FD stands for FunctionDefinition, and ES stands for ExpressionStatement.

C. Pattern Matching

GoHigh searches the AST extracted in the previous step for known patterns of low-level functions that we manually extracted from the dataset. To extract the patterns of low-level functions, we first find all possible representations of DS and constant string LC (as mentioned in Section III-D, we focus on these two categories of low-level functions). We present examples of these representations in Table II. We find that there are two methods to encode constant string LCs in our dataset. The first method, "direct encode", encodes it manually by calculating the keccak256 digest of the function name, and truncating the bytes after the 4

TABLE II: The representations of DS and constant string LC.

Category	Name	Example
DS	Direct	`addr.send(10ether);`
Constant String LC	Direct Encoded	`addr.call(bytes4(bytes256(keccak256("register(string)", "John"))));`
Constant String LC	ABI Encoded	`addr.call(ABI.encodeWithSignature("register(string)", "John"));`

leading bytes. The second method invokes a built-in function, "ABI.encodeWithSignature", and thus we call this method "ABI encode". Solidity also provides other encoding functions, *e.g.* , "ABI.encode" and "ABI.encodeWithSelector"; however, only "ABI.encodeWithSignature" encodes the function signature, while the others work with a short byte format of function signature (known as "selector" in Solidity). On the other hand, there is only one method for DS, and we call this "Direct".

Second, we iteratively distill the patterns in our dataset of DSs and constant string LCs. For each contract, we write a regular expression to match the low-level functions. The regular expression is as narrow as possible to avoid capturing other patterns. We then remove all contracts that match the regular expression, and repeat this process until there is no contract left. Thus, we formulate regular expressions for the patterns of low-level functions.

Finally, we inspect the above regular expressions to condense them into a smaller set. To do so, we develop ASTs for the patterns distilled in the previous step. We then identify similar AST pairs and attempt to merge them into a new AST. For example, the second example shown in the Require pattern in Table III adds a string node (message) to the first example's AST. We group these two patterns together due to their similarity. We iteratively repeat the process until we converge. Finally, we obtain the patterns shown in Table III. GoHigh looks for these patterns in the AST of the extracted contract, and performs targeted replacements of the same.

D. Replacement

Table III shows the replacement for each pattern identified in the previous section, in terms of the AST transformation. Each pattern has a custom replacement based on its AST. For example, though the if-clause pattern and the if-not pattern both protect the statements located in the if block, they require different replacement patterns as their behaviors differ.

Fig. 8a shows a typical example of if-not pattern, while Fig. 8b shows an example of if-clause pattern. Both examples call the "register" function and roll back if the call fails. In addition, the if-clause pattern performs extra operations (*e.g.* , sending recipient to the caller) in the if statement, which must be preserved after the replacement. Therefore, we keep the if statement in the if-clause pattern, and only replace the expression with a Boolean value True. The call to register is moved to the line before the if statement. However, in the if-not pattern, the operations after a successful call are located outside of the body of the if clause. Thus, there is no need to preserve any statements after the replacement.

To perform the replacement, we perform a level order traversal of the AST, which is a breath-first traversal of the tree. For each expression node, the function iteratively replaces the node or the sub-tree with the high-level expression node until it reaches the leaf node.

```
1 if(!address(0x123).call(abi.encodeWithSignature(
2     "register(string)", "John"))){
3     revert();
4 } //before
5 address(0x123).register("John"); //after
```

(a) The If-not pattern.

```
1 if(address(0x123).call(abi.encodeWithSignature(
2     "register(string)", "John"))){
3     send_receipt();
4 } else{
5     revert();
6 } //before
7 address(0x123).register("John");
8 if(True){send_receipt();} else{revert();} //after
```

(b) The If-clause pattern.

Fig. 8: Difference between if-not and if-clause pattern.

E. Implementation

We implemented GoHigh using a JSONPath-NG package, and SIF [16], a framework of contract instrumentation, to decompile AST form of the contract into its source code form. We have made *GoHigh's source code publicly available*[8].

F. Example

We provide an example to demonstrate how our replacement works - the source code is shown in Fig. 9. The example is extracted from a real-world contract called the Buttonwood Agreement (BATT) [9], which has more than $1,000$ transaction records in Ethereum. The example contains a contract with one vulnerable function. The function `approveAndCall` can increase a user's allowance and send it a message using an external function, whose address and function name are hard coded into the `_spender` variable (Line 5) and the payload (Line 6). Then, an LC is used in the require statement in Line 6. At the end of the function, the return statement returns True if the LC is correctly invoked. Thus, there is a constant string LC in Line 6 of the contract.

The AST of the example is shown in Fig. 10a. The root node is the StandardToken contract (Source). It has only one child node, approveAndCall (FunctionDefinition, FD), because it has only one member function. There are four children of the function declaration: value assignment node (VA), approval event node (AE), require statement node, and return node.

After obtaining the AST, we traverse it from the root node, and check if the tree matches the patterns listed in

[8]https://github.com/DependableSystemsLab/GoHigh.

[9]address: 0x2a6a1521a43601c847dd853cbc5f25d5d6505dad

TABLE III: The patterns of DS and LC for replacement, and the percentages of their occurrence in the base dataset (they do not add up to 100% as some contracts have multiple patterns). LL means low-level function; HL means high-level alternative.

Exists In	Name	%age	Example	AST Representation	Replacement
Both	Stand Alone	21.7%	`address(0x123).send(10ether);` `address(0x123).call(ABI.encodeWithSignature("register(string)", my_name));`	FD → ..., LL	FD → ..., HL
Both	If Clause	41.1%	`if(address(0x123).send(10ether)){...}` `if(address(0x123).call(ABI.encodeWithSignature("register(string)", my_name))){...}`	FD → ..., if → LL, block	FD → ..., HL, block
DS Only	If Not Clause	15.4%	`if(!address(0x123).send(10ether)){revert();}` `if(!address(0x123).send(10ether)){throw;}`	FD → ..., if → cond → NOT → LL, block	FD → ..., HL

DS Only	Require	26.3%	`require(address(0x123).send(10ether));`
			`require(address(0x123).send(10ether), "ERROR_MESSAGE");`
			`assert(address(0x123).send(10ether));`
				FD → ..., require - LL	FD → ..., HL

Both	Return	2.4%	`return address(0x123).send(10ether);` `return address(0x123).call(ABI.encodeWithSignature("register(string)", my_name));`	FD → ..., return – LL	FD → ..., HL, return – True

```
contract StandardToken is Token {
    function approveAndCall(address _spender,
        uint256 _value, bytes _extraData)
        returns (bool success) {
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
        require(_spender.call(
            bytes4(bytes32(sha3("receiveApproval(
                address,uint256,address,bytes)")))
                , msg.sender, _value, this,
                _extraData));
        return true;}}
```

Fig. 9: The example before replacement.

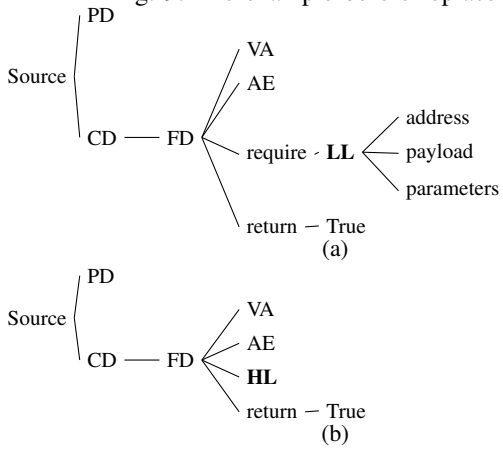

Fig. 10: Example's AST (a) before replacement, and (b) after replacement. HL stands for high-level function.

Table III. In the example, the require statement matches the require pattern of constant string LC. Therefore, GoHigh extracts the address (`_spender`), the payload (`"receiveApproval(address,uint256,address,bytes)"`) and the parameter list (`msg.sender, _value, this, _extraData`) of the low-level function. The extracted components are then reconstructed into a new statement that uses high-level function as per Table III. After replacement, the AST of the example is shown in Fig. 10b. Finally, the replaced AST is converted to the source code shown in Figure 11. The high-level alternative is in Line 5.

Note that the replacement leads to a gas cost reduction during contract execution. The original version shown in Fig. 9 costs 24,491 gas, while the replaced version in Fig. 11 costs only 24,262 gas, which is 229 gas less than that the original.

```
contract StandardToken is Token {
    function approveAndCall(address _spender,
        uint256 _value, bytes _extraData) returns
        (bool success) {
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
        contract_wrapper(_spender).receiveApproval(
            msg.sender, _value, this, _extraData);
        return true;}}
contract contract_wrapper{
    function receiveApproval(address a,uint256 b,
        address c,bytes d);}
```

Fig. 11: The example after replacement.

V. Experiment

A. Experimental Setup

1) Datasets: We used two non-overlapping datasets for our experiments, as shown in Table IV [10]. The first is the dataset that we analyzed in Section III, which has all the contracts on Ethereum from March 2016 to September 2019. We refer to this as the "base dataset". The second dataset is obtained by extracting the Ethereum contracts created between October 1, 2019, and January 1, 2021. We refer to this dataset as "recent dataset". This dataset consists of 170,304 contracts whose source code is available on Etherscan. *Note that the*

[10]We have made both datasets publicly available in [17].

contracts in the recent dataset were released almost a year after the Solidity guidelines were finalized in November 2018.

TABLE IV: The datasets used in the experiments.

Name	Date	Compiler Versions	No. of contracts
Base	Mar,2016 - Sep,2019	0.4.0-0.5.4	149, 363
Recent	Oct,2019 - Jan,2021	0.4.16-0.7.2	170, 304

We also collect the Ethereum transactions on the set of contracts used in both datasets. We collect all the transactions from September 2016 to January 2021 whose destination address matches a contract's address in either dataset. These transactions are used to determine if the replacement by GoHigh had any unintended changes to the contract's behavior.

2) Method and Metrics.: We run all experiments on an Intel Core i7-7700 3.6GHz machine with 32 GB of RAM running Ubuntu 20.04 LTS. The Solidity compilers we used are downloaded from the official website[11]. The private Ethereum blockchain node used for testing uses Geth 1.10.9-stable. The EVM version is Byzantium.

3) Research Questions (RQs).: We pose five RQs that we attempt to answer with our evaluation:

- **RQ1**: How do the characteristics of the recent dataset in terms of low-level functions compare to the base dataset?
- **RQ2**: What is the coverage of GoHigh in replacing the low-level functions in both datasets?
- **RQ3**: How many compiler warnings are eliminated in the contracts due to GoHigh's replacement?
- **RQ4**: Does the replacement by GoHigh introduce any unintended side-effects in the contracts?
- **RQ5**: What's the performance overhead of GoHigh?

To answer **RQ1**, we repeat the analysis in Section III on the recent dataset, and compare the results with those we obtained from the base dataset. As before, we considered both the overall contracts and the unique contracts in both datasets.

To answer **RQ2**, we first run GoHigh on the contracts, and then run a substring matching script (derived from Table II) on the contracts replaced by GoHigh to determine how many low-level functions are still left in the contracts after replacement. We measure the coverage of GoHigh on both datasets.

To answer **RQ3**, we compile the contracts after they have been replaced with GoHigh in both datasets. We then collect the compiler warnings, and determine if there are differences between these and the warnings collected when compiling the original contracts (we remove the original contracts that do not compile even before replacement from the datasets).

To answer **RQ4**, we first extract the public variables of each contract (i.e., variables declared with either the 'public' modifier or no modifier) along with its balance to determine the state of the contract. Then, we deploy both the original and replaced contracts on a private Ethereum blockchain node, after removing the original contracts that fail to deploy in our node. Finally, we replay the transactions in the transaction logs, and compare the external states of the contracts with each other. We say a replacement has 'succeeded' if the states match each other after the replay, as this suggests that no unintended side-effect was introduced by GoHigh. We measure the success rate for both datasets.

Finally, to answer **RQ5**, we measure the minimum, maximum, and average times taken by GoHigh for both datasets.

[11]https://binaries.soliditylang.org/

B. Results

We organize the results by the RQs.

RQ1 (Observations from recent dataset). We observe that the percentage of contracts that contain low-level functions in the recent dataset is 37.8% (shown in Fig.12a), which is 24.0% higher than that the base dataset. The percentage of unique contracts also increases from 17.2% in base dataset to 23.5% in the recent dataset, as shown in Fig.12b. *This suggests that low-level functions are actually increasing in number in the recent dataset despite the publication of the guidelines.*

Fig.13 shows the distribution of the low-level functions in the recent dataset. The most significant change is that the major category changes from constant string LC to DS in the recent dataset (at 71.7%). An in-depth analysis of the contracts in the recent dataset reveals that one-fourth of the contracts are Forwarder contracts, which serve as a payment forwarder to a user's wallet, and contain DSs. Further, many of the contracts are proxy contracts. A proxy contract uses arbitrary LCs to dynamically invoke functions [18]. It is used to upgrade contracts as it is not possible to upgrade a smart contract once it is deployed on the blockchain [19]. This is why DS and arbitrary LC are more prevalent than constant string LC in the recent dataset. *With the above said, GoHigh can still replace more than 75% of the low-level functions in this dataset (*71.7%*DS* + 4.9%*CSLC* + 0.4%*Other* = 77.0%*).*

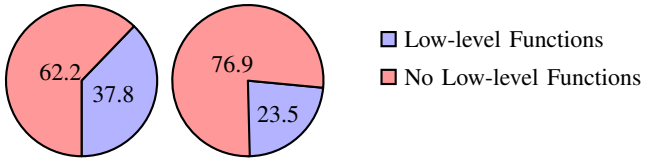

Fig. 12: Low-level functions usage in the recent dataset for (a) all contracts and (b) unique contracts.

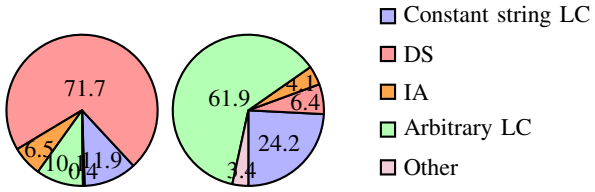

Fig. 13: The distribution of low-level functions in the recent dataset for (a) all contracts and (b) unique contracts.

RQ2 (Coverage of GoHigh's replacement)

All the DS and constant string LC contracts in the both the base and recent datasets are correctly identified by GoHigh. After replacement by GoHigh, none of the target contracts contain either DS or constant string LC low-level functions. While this result is not surprising for the base dataset (as we used it to identify the replacement patters in Table III), the recent dataset was also completely covered by the patterns

identified from the base dataset despite it not being used in pattern extraction. *Therefore, GoHigh has an overall coverage of 100% in identifying the patterns of low-level functions in both the datasets*. Recall that this constitutes 82% of the contracts using low-level functions in the base dataset (Section III), and 77% in the recent dataset (RQ1).

Note that we observed that six contracts in the recent dataset used "send" even after replacement. However, these are not missed cases because the "send" function is overridden in their code, and thus no longer belongs to the DS category.

RQ3 (Compiler warnings before and after replacement) In the base dataset, all contracts passed compilation after replacement by GoHigh (*i.e.*, there was no contract that did not compile after the replacement). Figure 14 shows the changes in warnings for contracts using DS and constant string LC before and after replacement in the base dataset. As can be seen, the majority of compiler warnings are unchanged as they do not pertain to low-level functions. Further, there is a 16% reduction in warnings due to the replacement of low-level functions (shown as 'Remove' in the figure). For example, the compiler expects developers to check the return value of LCs, and will output a warning if the check is missing. Because GoHigh replaces LC, the compiler will not emit this warning. Similarly, using DS anywhere in the contract results in a warning. GoHigh removes all DSs in contracts, and so the warnings due to the use of DS are also removed.

However, a few new warnings are added due to the replacement of low-level functions in some cases (shown as 'Add' in the figure). Most of these are due to unused variables after the replacement. For example, in Fig. 15, replacing the LC does not remove the declaration of the Boolean variable `result` in Line 5. However, these are benign warning as they do not affect either the contract's correctness or its security.

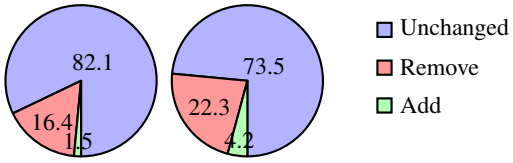

Fig. 14: Change in compiler warnings for the base dataset for (a) DS, and (b) Constant String LC.

```
1 bool result;
2 result = address(0x123).call(
3     abi.encodeWithSignature(
4     "register(string)", "John")); //before
5 bool result; //warning: Variable declared but
      unused
6 address(0x123).register("John"); //after
```

Fig. 15: Warning Added after replacement by GoHigh.

In the case of the recent dataset, 99.9% of the DS contracts and 95.4% of constant string LCs successfully passed compilation after replacement by GoHigh. It is not 100% due to a recently introduced modifier in Solidity regarding the storage location of dynamic-sized arrays (*e.g.* , string, byte array). Our analysis does not currently handle this modifier, and hence the replacement by GoHigh results in compiler errors. Extending our analysis for this case is a potential future work avenue.

The results of the changes in compilation warnings for the recent dataset are shown in Fig.16. As can be seen, the reductions in warnings for contracts using deprecated send is consistent with the base dataset, *i.e.*, 17% warning decrease. However, there is a 38.2% decrease in the compiler warnings in the constant string LC contracts, which is 16% higher than the decrease in warnings in the base dataset.

Overall, GoHigh significantly decreases the number of compilation warnings in the smart contracts after replacement.

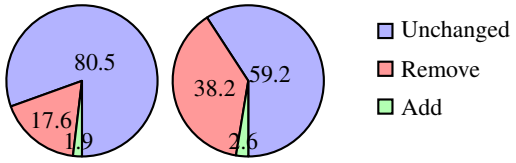

Fig. 16: Change in compiler warnings for the recent dataset for (a) Deprecated Send (DS), (b) Constant String LC.

RQ4 (Unintended Side-Effects After Replacement) For the base dataset, we find that the success rate of GoHigh is 98.7% and 94.1% on contracts using DS and constant string LC, respectively. *The overall success rate of GoHigh for the base dataset is* 95.1%. The success rate of GoHigh in the recent dataset is 68.6% on contracts using deprecated send, and 96.5% on contracts using constant string LCs. *The overall success rate of GoHigh is* 92.4% *for the recent dataset.*

The reason why some contracts do not succeed is that their transactions use arbitrary calls to functions defined in external contracts. We cannot confirm that the state matches for these contracts if the external contracts modify the state of the tested contracts in arbitrary calls, as we do not deploy the external contracts on our private blockchain (because it is non-trivial to decode their address based on the call and to replicate them). Note however that just because a contract did not succeed does not mean that the replacement done by GoHigh is incorrect, but rather that we are unable to verify its correctness. *In other words, none of the contracts in which GoHigh performed replacements failed to match the states of the original contracts, for the replayed transactions.*

Overall, GoHigh was able to replace low-level functions in the vast majority of contracts (over 92%) without any externally visible state changes, in both datasets.

RQ5 (Performance of GoHigh). The average time taken by GoHigh, including the AST generation, is 6.59 seconds across both datasets. The minimum time is 0.11 seconds, and the maximum is 120 seconds. Finally, GoHigh had a replacement time of under 60 seconds for 97.09% of the contracts.

VI. Threats to Validity

There are four threats to the validity of the experiments.

1. An *External* threat to validity is dataset sampling bias as we only collected open-source contracts to perform the evaluation experiments. The total number of contracts submitted to Ethereum is 17 million, but our base dataset has just under 150,000 contracts, which is less than 1% of all the contracts

in Ethereum. However, we do not consider the other contracts as it is non-trivial to determine whether a contract uses a low-level function without having access to its source code. Further, this problem is not unique to GoHigh; for example, prior work SMARTSHIELD [20] has the same issue.

2. An *Internal* threat to validity is the difference in the versions of Ethereum Virtual Machine (EVM) used for running the experiments. We run all the experiments on the default EVM version of `solc` compiler, Byzantium. However, some errors in deployment resulted from the mismatch of EVM version, as the EVM versions of the contract vary. This issue can be alleviated by using different versions of the EVM, but is logistically more challenging to deploy.

3. Another *Internal* threat to validity is that we do not handle all IA categories. Although we included IA in Section II, we currently do not have a systematic understanding of developers' use of this category. This is because there is wide variation among the different uses of this category, and we hence cannot distill patterns from the datasets to capture IAs.

4. Another *External* threat to validity are external call transactions, which make it infeasible for our verification process to check whether the state changes are the same for the replaced contract as the original one. This is because we run the transactions on a private blockchain, as we do not want to pay gas for running them on the public blockchain. This can be alleviated by extracting these contracts and running them on the private blockchain, but requires manual effort.

VII. Related Work

We classify related work into two broad categories.

Smart Contract Vulnerability Detection There has been significant work in detecting vulnerabilities and software bugs in smart contracts [4], [8], [9], [21]–[33]. Previous studies investigated various aspects of smart contract vulnerabilities (*e.g.* , airdrop hunting [34] and honeypot smart contract [35]), and proposed tools for protecting smart contracts from these vulnerabilities, *e.g.* , providing smart contract weakness classification and test cases, static analysis and dynamic analysis.

However, only a few papers have considered patching the discovered vulnerabilities. SMARTSHIELD [20] was the first tool on protecting low-level functions by adding a return value check to each of them. However, SMARTSHIELD has three shortcomings. First, a simple return value check does not address the issue if the callee's address does not exist - the function call will return `True` even though it fails. Second, SMARTSHIELD operates on the smart contract's source code, but only outputs its patched bytecode. This makes it difficult for developers to audit the check. Third, SMARTSHIELD relies on existing tools (Securify [4], Osiris [24], Mythril [9]) to label the insecure cases, however, the detection rate of these tools is quite low as prior work has found [36].

Another recent technique, EVMPatch [10] protects not only the low-level function statement but the whole function by adding access control to it. The access controls check whether a contract is properly initialized before being used (whether the owner has a valid address), and restrict the caller of sensitive and vulnerable functions (*e.g.* initialize, self-destroy, functions that contain low-level functions). However, the patches only hide the vulnerability behind the access control mechanism instead of addressing its root cause. The low-level functions are still present, and can be exploited if the access control is not correctly implemented.

Furthermore, both SMARTSHIELD and EVMPatch require significant intervention by developers in writing vulnerability detection instructions in the bytecode level, unlike GoHigh, which is completely automated.

Insecure Feature in Other Languages Insecure features in other programming languages have been well-studied. For example, Richards *et al*. performed a large-scale study on the usage of `eval` in JavaScript [37]. Prior to that work, they carried out a more general analysis on more dynamic features of JavaScript [38]. The low-level functions in Solidity are similar to the `eval` call in JavaScript, which allows developers to convert strings into executable code. Similarly, Holkner and Harland [39] found that dynamic code execution is also widely adopted in Python. Livshits *et al*. [40] conducted static analysis on Java reflection, including reflective calls whose functionality is similar to Solidity LCs. Insecure calls in C code are frequent as well, e.g., Austin *et al*. proposed a source-to-source code transformation tool to address the point and array access errors, and LibSafePlus [41] provides runtime protection against buffer overflows. However, the low-level functions in Solidity are fundamentally different.

VIII. Conclusion

We defined low-level functions that lead to security vulnerabilities in Ethereum smart contracts written in the Solidity language. We carried out a large-scale empirical study of the use of low-level functions in Ethereum smart contracts. We found that low-level functions are widely used (at 13.8%). The number increased to 37.8% when we analyzed a recent dataset consisting of contracts deployed well after the guidelines had been finalized. Further, we find that more than 75% of the low-level functions are gratuitous for the contract's functionality.

We proposed GoHigh, an automated source-to-source transformation tool for replacing low-level functions in smart contracts with their high-level alternatives. We evaluated GoHigh on both datasets consisting of over $300,000$ real-world contracts deployed on Ethereum. The results show that GoHigh is able to *replace all the contracts that contain low-level functions* that are amenable to be replaced, and takes an average time of seven seconds per contract. Further, GoHigh reduces over 17% of compiler warnings after replacement, and achieves over 92% success rate in matching the external state changes of the contracts before and after the replacement.

Acknowledgements

This work was funded in part by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC). We also thank the anonymous reviewers of SANER'22 for their insightful comments.

REFERENCES

[1] G. A. Pierro, R. Tonelli, and M. Marchesi, "An organized repository of ethereum smart contracts' source codes and metrics," *Future internet*, vol. 12, no. 11, p. 197, 2020.

[2] M. I. Mehar, C. L. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H. M. Kim, and M. Laskowski, "Understanding a revolutionary and flawed grand experiment in blockchain: the dao attack," *Journal of Cases on Information Technology (JCIT)*, vol. 21, no. 1, pp. 19–32, 2019.

[3] S. Palladino. (2017) The parity wallet hack explained. [Online]. Available: https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7

[4] 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.

[5] L. Marchesi, M. Marchesi, L. Pompianu, and R. Tonelli, "Security checklists for ethereum smart contract development: patterns and best practices," *arXiv preprint arXiv:2008.04761*, 2020.

[6] Z. Wan, X. Xia, D. Lo, J. Chen, X. Luo, and X. Yang, "Smart contract security: a practitioners' perspective," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1410–1422.

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

[8] 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.

[9] ConsenSys. (2018) Mythril github repository. [Online]. Available: https://github.com/ConsenSys/mythril

[10] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Evmpatch: timely and automated patching of ethereum smart contracts," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[11] E. Lai and W. Luo, "Static analysis of integer overflow of smart contracts in ethereum," in *Proceedings of the 2020 4th International Conference on Cryptography, Security and Privacy*, 2020, pp. 110–115.

[12] L. Alt and C. Reitwiessner, "Smt-based verification of solidity smart contracts," in *International Symposium on Leveraging Applications of Formal Methods*. Springer, 2018, pp. 376–388.

[13] A. Hefele, U. Gallersdörfer, and F. Matthes, "Library usage detection in ethereum smart contracts," in *OTM Confederated International Conferences" On the Move to Meaningful Internet Systems"*. Springer, 2019, pp. 310–317.

[14] G. Ayoade, E. Bauman, L. Khan, and K. Hamlen, "Smart contract defense through bytecode rewriting," in *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, 2019, pp. 384–389.

[15] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "Easyflow: Keep ethereum away from overflow," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 23–26.

[16] C. Peng, S. Akca, and A. Rajan, "Sif: A framework for solidity contract instrumentation and analysis," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2019, pp. 466–473.

[17] R. Xi and K. Pattabiraman, "Gohigh's dataset," Jan. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.5843540

[18] openzeppelin. (2018) Proxy patterns. [Online]. Available: https://blog.openzeppelin.com/proxy-patterns/

[19] F. Hofmann, S. Wurster, E. Ron, and M. Böhmecke-Schwafert, "The immutability concept of blockchains and benefits of early standardization," in *2017 ITU Kaleidoscope: Challenges for a Data-Driven Society (ITU K)*. IEEE, 2017, pp. 1–8.

[20] Y. Zhang, S. Ma, J. Li, K. Li, S. Nepal, and D. Gu, "Smartshield: Automatic smart contract protection made easy," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 23–34.

[21] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

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

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

[24] 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.

[25] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018, pp. 1–5.

[26] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 454–469.

[27] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 1317–1333.

[28] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, pp. 1–27, 2018.

[29] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *Ndss*, 2018, pp. 1–12.

[30] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–28, 2017.

[31] J. Frank, C. Aschermann, and T. Holz, "{ETHBMC}: A bounded model checker for smart contracts," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2757–2774.

[32] 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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[33] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.

[34] 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 *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 2793–2810.

[35] C. F. Torres, M. Steichen *et al.*, "The art of the scam: Demystifying honeypots in ethereum smart contracts," in *28th {USENIX} security symposium ({USENIX} security 19)*, 2019, pp. 1591–1607.

[36] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 415–427.

[37] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do," in *European Conference on Object-Oriented Programming*. Springer, 2011, pp. 52–78.

[38] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010, pp. 1–12.

[39] A. Holkner and J. Harland, "Evaluating the dynamic behaviour of python applications," in *Proceedings of the Thirty-Second Australasian Conference on Computer Science-Volume 91*, 2009, pp. 19–28.

[40] B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for java," in *Asian Symposium on Programming Languages and Systems*. Springer, 2005, pp. 139–160.

[41] K. Avijit, P. Gupta, and D. Gupta, "Binary rewriting and call interception for efficient runtime protection against buffer overflows," *Software: Practice and Experience*, vol. 36, no. 9, pp. 971–998, 2006.