Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2021.DOI

Evaluating countermeasures for verifying the integrity of Ethereum smart contract applications

SUHWAN JI[1], DOHYUNG KIM[1,2], AND HYEONSEUNG IM[1,2],

[1]Major in AI and Software, Interdisciplinary Graduate Program in Medical Bigdata Convergence, Kangwon National University, Chuncheon 24341, South Korea
[2]Department of Computer Science and Engineering, Kangwon National University, Chuncheon 24341, South Korea

Corresponding authors: Dohyung Kim (e-mail: d.kim@kangwon.ac.kr) and Hyeonseung Im (e-mail: hsim@kangwon.ac.kr)

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2019R1F1A1063272, 2020R1F1A1048395, and 2020R1A4A3079947).

ABSTRACT Blockchain technology, which provides digital security in a distributed manner, has evolved into a key technology that can build efficient and reliable decentralized applications (called DApps) beyond the function of cryptocurrency. The characteristics of blockchain such as immutability and openness, however, have made DApps more vulnerable to various security risks, and thus it has become of great significance to validate the integrity of DApps before they actually operate upon blockchain. Recently, research on vulnerability in smart contracts (a building block of DApps) has been actively conducted, and various vulnerabilities and their countermeasures were reported. However, the effectiveness of such countermeasures has not been studied well, and no appropriate methods have been proposed to evaluate them. In this paper, we propose a software tool that can easily perform comparative studies by adding existing/new countermeasures and labeled smart contract codes. The proposed tool demonstrates verification performance using various statistical indicators, which helps to identify the most effective countermeasures for each type of vulnerability. Using the proposed tool, we evaluated state-of-the-art countermeasures with 237 labeled benchmark codes. The results indicate that for certain types of vulnerabilities, some countermeasures show evenly good performance scores on various metrics. However, it is also observed that countermeasures that detect the largest number of vulnerable codes typically generate much more false positives, resulting in very low precision and accuracy. Consequently, under given constraints, different countermeasures may be recommended for detecting vulnerabilities of interest. We believe that the proposed tool could effectively be utilized for a future verification study of smart contract applications and contribute to the development of practical and secure smart contract applications.

INDEX TERMS Blockchain, countermeasure, Ethereum, smart contract, vulnerability

I. INTRODUCTION

Since Bitcoin [1], which was designed using blockchain, was introduced, blockchain technology has evolved and interests in its applications have greatly been increasing. With the ability to provide digital security in a distributed manner, blockchain has been used to develop a variety of decentralized applications across the industry. In particular, such decentralized applications, called DApps, often operate upon the Ethereum Virtual Machine (EVM), and they are built using smart contracts which are a piece of code that enables DApps to interact with the underlying Ethereum blockchain [2].

Despite the great advantages of using the blockchain technology, however, it has ironically been revealed that smart contracts are vulnerable to various security risks due to the blockchain's essential features, such as transparency and immutability [3]–[6]. For example, if a smart contract incurs a wrong transaction (by mistake or malicious attacks) and the result is once written to the blockchain, then the transaction can hardly be corrected. Rather, the blockchain should be destroyed (or hardforked). Because of that reason, it is of considerable importance to test the integrity and safety of smart contract applications before they are actually used in conjunction with the blockchain.

As a result of recent research on vulnerabilities in smart contracts, representative vulnerabilities were introduced [7],

[8], and various countermeasures were proposed [5], [6], [8]–[10]. However, the effectiveness of the countermeasures has not been properly studied. Most performance evaluations have been performed using unlabeled data. Hence, their performance comparisons are often not conclusive since when using unlabeled data, even if a countermeasure detected some vulnerabilities, it is not clear whether they were actual bugs or false positives, and also how many vulnerabilities each countermeasure missed. For example, recently, the authors of [10] used 47,518 smart contracts for comparative studies, but reported the test results without confirming that vulnerabilities actually exist in the smart contracts. Only 69 contracts having 115 vulnerabilities in total were used as labeled data, resulting in that it is not clear what the most effective countermeasures are for each type of vulnerability. Besides, a comparative study itself may be cumbersome and time-consuming tasks since different countermeasures may require different environments to run, and the analysis results in different formats should be additionally arranged.

In this paper, a new software tool is designed to facilitate validation of the existing/new countermeasures, into which the user can easily add new countermeasures and labeled benchmark datasets. In particular, it automatically analyzes the results of executing the countermeasures on the available benchmark datasets, and shows their performance using tables and graphs under various performance measures to facilitate easy comparison. To this end, the proposed tool is implemented using OS-level virtualization and operates within a Docker container, allowing it to operate independently of the underlying system and eliminating the need for the user to perform separate installation/execution for each countermeasure. As a result, new countermeasures (as well as labeled benchmark smart contract codes) can easily be included and evaluated in the proposed tool, and their performance can effectively be cross-checked using various metrics. Using the proposed tool and 237 labeled smart contract codes, we evaluate the representative existing countermeasures in the literature.

The evaluation results show that, in general, the countermeasures identifying more vulnerable code produce a much larger number of false positives, resulting in very low precision and accuracy. The effectiveness of countermeasures against ‘Access Control’, ‘Denial of Service’, and ‘Front-Running’ are questionable. The F1-scores of all countermeasures are less than 25%. Vulnerable codes with ‘Integer Overflow/Underflow’ and ‘Timestamp Dependence’ can be completely detected. However, the performance of the countermeasures needs to be further improved in order to reduce false positives. As for the vulnerabilities of ‘Reentrancy’ and ‘Unchecked Low Level Call’, we confirm that there are effective countermeasures that show both high precision and high recall values. We believe that the proposed tool will contribute to a future verification study of smart contracts and development of practical and secure smart contract applications.

The main contributions are summarized as follows:

- The nine representative vulnerabilities discussed in the Decentralized Application Security Project (or DASP) Top 10 of 2018 [7] and their state-of-the-art countermeasures are revisited.
- The limitations of the current countermeasures are discussed from the perspective of practicality, and an effective software tool that can evaluate the performance of the countermeasures with great convenience is designed.
- The proposed tool is implemented using an OS-level virtualization technique, and is open to the public via https://github.com/93suhwan/uscv.
- The proposed tool eliminates the need to manage a separate installation/execution environment for each countermeasure and provides easy comparative analysis, helping to identify the most effective countermeasures for each type of vulnerability.
- Using the proposed tool and 237 labeled data, we conduct a comparative study for the representative existing countermeasures in the literature, and their performance is represented using various performance measures.

The rest of the paper is organized as follows. Section II introduces Ethereum smart contracts and their vulnerabilities. Section III summarizes the state-of-the-art countermeasures for the vulnerabilities of smart contracts. In Section IV, we introduce the design of the proposed tool and discuss the results for evaluating the performance of the existing countermeasures using the proposed tool. Finally, Section VI concludes the paper.

II. PRELIMINARIES

A. BLOCKCHAIN AND ETHEREUM SMART CONTRACTS

The first blockchain was introduced in 2008 by a pseudonymous person or group known as Satoshi Nakamoto [1]. Essentially, a blockchain is a list of blocks that record information. Since blocks in the chain are connected using a cryptographic hash (more specifically, in the way that each block contains the cryptographic hash of the previous block in the chain), any information of a block in the chain can be changed only if all of its subsequent blocks can also be modified. However, since such modifications require the consent of the majority of the network, consequently, malicious change in the blockchain is almost impossible.

Since a blockchain can work as a distributed, verifiable public ledger that records transactions, its first application was a cryptocurrency, named Bitcoin [1]. In order to add a new block to the blockchain, Bitcoin uses a consensus mechanism called Proof-of-Work (PoW), where nodes in the network compete for generating a right block by solving a cryptographic puzzle. Extended from Bitcoin, Ethereum allows to store computer code that can be used to implement unforgeable decentralized applications [2], which is now being a building platform for running various kinds of DApps.

A smart contract is a piece of code that enables DApps to interact with the blockchain, and it actually runs on a quasi-Turing-complete virtual machine, called EVM. EVM

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3091317, IEEE Access

is considered as a sort of distributed machine that executes smart contracts that embed the DApp logic by consuming Ether (Gas in EVM). Since the blockchain has a property of immutability, once a smart contract is deployed on the blockchain, it cannot be modified like other transactions. Therefore, it is significant to test the integrity and safety of smart contracts before they are actually used upon the blockchain. Otherwise, the blockchain must be destroyed or hardforked if serious errors in the deployed smart contracts are found afterwords.

B. VULNERABILITIES OF SMART CONTRACTS

This section briefly reviews the nine representative vulnerabilities discussed in the DASP Top 10 of 2018 [7].

- Reentrancy (V_{re}): Before a contract is completed (i.e., resolving any effects), the contract is executed recursively or other contracts are invoked to make the state in a mess. Below is an example scenario that exploits a reentrancy vulnerability.

```
function withdraw(){
 - Transfer tokens to someone.
 - Update balance.
}

1. The attacker invokes the function
   withdraw in succession.
2. The second function call is done
   before balance has not been updated
   for the first function call.
3. balance is updated only for
   the second function call.
```

- Access Control (V_{ac}): Contract's private values or functions are accessed abnormally due to an insecure visibility setting. Below is an example that describes an access control vulnerability. This function does not check whether the function was already called and the state has already been initialized.

```
function initState(){
   owner = msg.sender
}

1. The function can be called abnormally
   via a delegatecall.
2. Then, the value of owner could
   be manipulated.
```

- Integer Overflow/Underflow (V_{io}): Solidity uses variables of unsigned int type. If programmers process variables of unsigned int type as if they were the variables of signed int type, an overflow and underflow can occur. If such errors happen, for example, a wrong amount of tokens can be withdrawn. Below is an example that shows an integer underflow.

```
function withdraw(uint amount){
  if(balance - amount > 0){
    - Withdraw tokens.
  }
}

1. Suppose balance = 0 and amount = 1.
2. The value of (balance - amount) can
   be interpreted positive since balance
   is a value of unsigned int.
```

- Unchecked Low Level Call (V_{uc}): When errors happen in low level functions in Solidity, a boolean value set to false is returned, but the code keeps running. Therefore, the result of such low level functions should be checked to confirm successful execution. Below is an example that shows an unchecked low level call vulnerability.

```
function withdraw(uint amount){
 - balance is updated
   (i.e., balance -= amount).
 - Transfer tokens (as many as amount)
   by calling a send function.
}

1. If send function call fails, balance
   is managed incorrectly.
```

- Denial of Service (DoS, V_{dos}): When DoS attacks are launched, smart contracts can be unavailable. Various types of DoS implementation have been reported including increasing gas necessary, abusing access control, and maliciously behaving. Below is an example that shows a sort of DoS. Computation at each block is limited by the upper bound of the amount of gas in Ethereum. If the function (doSomething), called by the attacker, has a heavy code that consumes too much gas, other transactions cannot be included in the block.

```
function doSomething(){
  for(uint i = 0; i < N; i++){
    - Heavy code.
  }
}

1. Attackers call doSomething.
2. Too much gas is consumed using heavy
   code in doSomething.
3. Other transactions cannot be included
   in the block since the gas limit for
   the block is reached.
```

- Bad Randomness (V_{br}): Generation of a random number is required in several applications such as games and lotteries. However, it is tricky to implement a random number generation on the Ethereum public blockchain since 1) Ethereum is a deterministic Turing machine without embedding true randomness, and 2) all the data (block variables) used for generating a random number is open to public, even to attackers. Hence, an attacker can predict the sources of randomness to some extent and replicate it to attack the function relying on the random value. Obviously, instead of using block variables open to the public, a random value can be created using timestamps. However, as discussed below for timestamp dependence, the timestamps can be manipulated by miners, resulting in another type of attacks. Below is an example that exploits a bad randomness vulnerability.

```
function coinFlip(bool guess){
  value = a hash value generated using
          a block number
  side = value / given denominator
  if (side == guess){
    - Win the game.
  }
}

1. Attackers can always win using the
   function exploit below.
2. Copy the function coinFlip and get
   the result in advance (A).
3. Call the function coinFlip based on
   the result (B).

function exploit(bool guess){
  (A)
  value = a hash value generated using
          the same block number
  side = value / given denominator

  (B)
  if(side == guess){
    conFlip(guess);
  } else{
    conFlip(!guess);
  }
}
```

- Front-Running (V_{fr}): Miners perform calculation while being compensated for the gas. The more gas (higher fees), the more quickly the transactions can be computed. Since the public Ethereum is transparent, pending transactions are visible to anyone. Hence, attackers can preempt the results of an already calculated transaction by copying the transaction at a higher fee. Below is an example scenario that exploits a front-running vulnerability.

```
1. Information sent in a transaction $T_a$
   (Ether, recipient address) is public.
2. Time elapses until $T_a$ is confirmed.
3. $T_a$ is read by an attacker before it
   is confirmed.
4. The attacker's transaction $T_b$, which
   is generated by copying $T_a$, is placed
   before $T_a$.
5. The attacker can steal the result of
   computing $T_a$.
```

- Timestamp Dependence (V_{td}): The timestamp of a block is determined by the miner (they reports the time at which the mining occurs). However, it can be manipulated by the miner (the timestamp can be changed within 15 seconds). Hence, fake time can be advertised by malicious miners, which allows the output of the contract to be changed.

- Short Address (V_{sa}): When a contract receives data of smaller-than-expected size, the missing portion is padded to zeros in EVM. For example, if the user address, signature, and the amount of token to be withdrawn are 0x12345600, 0xabcdef12, and 32(0x00000020), respectively, EVM concatenates all the values in the order of signature, address, and token amount, resulting in 0xabcdef121234560000000020. If an attacker specifies a short address such as 0x123456 instead of 0x12345600, in the previous case, EVM generates a value of 0xabcdef1212345600000020 and two zeros are padded at the end, resulting in 0xabcdef1212345600000020**00**. The resulting value can be misinterpreted as having to withdraw as many tokens as 0x00002000.

III. COUNTERMEASURES USING STATIC AND DYNAMIC ANALYSIS

In this section, we briefly examine 11 publicly available, open-sourced, representative countermeasures for Ethereum smart contracts with a command-line interface (CLI). Table 1 summarizes the characteristics of the considered countermeasures such as the main methods, input, and DASP Top 10 vulnerabilities supported. Among the main methods used by the countermeasures, static analysis refers to any kind of methods for examining and analyzing the code without actually executing it, whereas dynamic analysis refers to those for testing and evaluating the code by running it with test cases. Typical static analysis includes abstract interpretation, control-flow analysis, data-flow analysis, symbolic execution, etc., whereas dynamic analysis includes code coverage, memory error detection, fault localization, security analysis, etc. Static analysis is faster but less precise than dynamic analysis. In addition, static analysis finds properties that hold for all execution paths, whereas dynamic analysis finds those for one or more execution paths, but can detect subtle or complex vulnerabilities that static analysis may not detect. Below we review each countermeasure in alphabetical order of their names.

A. ECHIDNA

Echidna [11], [12] is an open-source, easy-to-use, property-based fuzz testing tool for Ethereum smart contracts, developed and used by Trail of Bits. Instead of using a predefined set of rules to detect vulnerabilities, it supports user-defined properties for property-based testing [30], arbitrary

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3091317, IEEE Access

TABLE 1: Overview of the countermeasures considered in our proposed tool. We considered only publicly available, open-sourced countermeasures with a CLI. **Year** denotes the publication year of the first relevant conference, workshop, or journal paper, if any. **Vulnerabilities** denote either those that can be detected by the given countermeasure (that is, the countermeasure implements a detector for the specified vulnerability) or its functionalities if the countermeasure is a testing tool, linter, or profiler. V_{ac}: Access control; V_{dos}: Denial of service; V_{fr}: Front-running; V_{io}: Integer overflow/underflow; V_{re}: Reentrancy; V_{td}: Timestamp dependence; V_{uc}: Unchecked low level call.

Countermeasure	Year	Main Methods	Input	Vulnerabilities
Echidna [11], [12]	2020	Property-based fuzzing	Solidity	User-defined property, assertion checking, gas use estimation
Ethlint [13]	None	Predefined and user-defined style and security rules	Solidity	Predefined and user-defined properties
Manticore [14], [15]	2019	Symbolic execution, dynamic analysis, SMT solving	Solidity	$V_{ac}, V_{fr}, V_{io}, V_{re}, V_{td}, V_{uc}$
Mythril [16], [17]	2018	Symbolic execution, SMT solving, taint analysis	Solidity EVM bytecode	$V_{ac}, V_{dos}, V_{fr}, V_{io}, V_{re}, V_{td}, V_{uc}$
Oyente [18], [19]	2016	Symbolic execution, SMT solving	Solidity EVM bytecode	$V_{ac}, V_{fr}, V_{io}, V_{re}, V_{td}$
Securify [20], [21]	2018	Static analysis, data-flow analysis, control-flow analysis	Solidity EVM bytecode	$V_{ac}, V_{fr}, V_{re}, V_{td}, V_{uc}$
Slither [22], [23]	2019	Data-flow analysis, taint traking	Solidity	$V_{ac}, V_{re}, V_{td}, V_{uc}$
SmartCheck [24], [25]	2018	Static and syntactic analysis	Solidity	$V_{ac}, V_{dos}, V_{io}, V_{re}, V_{td}, V_{uc}$
Solhint [26]	None	Predefined and user-defined sytle and security rules	Solidity	V_{re}, V_{td}, V_{uc}
Sol-profiler [27]	None	Syntactic analysis	Solidity	Predefined properties
VeriSmart [28], [29]	2020	CEGIS-style verification, SMT solving	Solidity	V_{io}

assertion checking, and estimation of maximum gas usage. That is, it automatically generates tests to detect violations in user-defined properties and assertions, and allows us to prevent vulnerabilities caused by out-of-gas conditions. Echidna uses the Slither static analysis tool [22], which we discuss below, in the preprocessing step to compile and analyze smart contracts and use information from Slither to improve fuzz testing. Currently, Echidna can also test contracts compiled with Vyper (https://vyper.readthedocs.io/en/stable/) and supports smart contract development frameworks such as Truffle (https://www.trufflesuite.com/) and Embark (https://framework.embarklabs.io/).

B. ETHLINT

Ethlint [13], formerly known as Solium, is a customizable, stand-alone linter for Solidity smart contracts. It provides a predefined set of various style and security rules, which the user can configure, for example, by choosing which rules to apply to the code or by passing options to the rules to modify their behavior. Ethlint was originally designed to strictly adhere to the Solidity style guide (https://solidity.readthedocs.io/en/develop/style-guide.html), but now it allows the user to not only customize the predefined rules but also write and distribute via NPM new plugins for their own rules. It can also automatically fix the detected style and security issues, but there is no benchmark result.

C. MANTICORE

Manticore [14], [15] is an open-source dynamic symbolic execution framework not only for Ethereum smart contracts but also for native binaries. It consists of the Core Engine implementing a generic platform-independent symbolic execution engine, the Native and Ethereum Execution Modules for symbolic execution of binaries and smart contracts, respectively, and the Satisfiability Modulo Theories (SMT) module and a Python API for supporting a customized analysis and interacting with external solvers such as Z3 (https://github.com/Z3Prover/z3), Yices (https://yices.csl.sri.com/), and CVC4 (https://cvc4.github.io/). Currently, Manticore supports various built-in vulnerability detectors such as for problematic uses of delegatecall, integer overflows, reentrancy bugs, uses of potentially insecure instructions, reachable external calls, reachable selfdestruct instructions, uninitialized memory and storage usage, invalid instructions, and unused internal transaction return values. The main downside of using Manticore is its long execution time; it is very much slower than other static analysis tools (while it took about 24 minutes on average, other tools just took from a few seconds to a few minutes under experiments using 47,518 contracts) [10].

D. MYTHRIL

Mythril [16], [17] is an open-source, interactive, security analysis tool for Ethereum smart contracts, which also supports other EVM-compatible blockchains such as Quorum (https://consensys.net/quorum/), VeChain (https://www.vechain.org/), and Tron (https://tron.network/). It is one of the earliest developed automated smart contract analysis tools and can be used to detect various security vulnerabilities such as use of delegatecall to untrusted contracts, integer overflows/underflows, and multiple sends in a single transaction. It uses various program analysis techniques such as symbolic execution, SMT constraint solving, taint analysis and control flow checking to detect such vulnerabilities. Mythril has been shown to be highly accurate in detecting the DASP Top 10 vulnerabilities when compared with other tools [9], [10]. It can also be used in a commercial SaaS smart contract security analysis platform called MythX (https://mythx.io/) which is more optimized and provides a wider range of functionalities.

E. OYENTE

Oyente [18], [19] is one of the first Ethereum smart contract analysis tools, which has served as a basis for the design and development of other tools such as HoneyBadger [31], Maian [32], and Osiris [33]. It performs symbolic execution and SMT constraint solving using the Z3 theorem prover to analyze EVM bytecode and detect various vulnerabilities. The authors of [18] conducted an experiment using existing 19,366 Ethereum smart contracts and reported that Oyente identified 8,833 contracts as vulnerable. However, several recent studies [9], [10] revealed that Oyente produces a considerable number of false positives, in particular, due to the integer overflow/underflow vulnerability, as is also discussed in Section IV-B. That is, Oyente is not appropriate for detecting arithmetic vulnerabilities. We also remark that while Oyente currently reports a call stack depth attack vulnerability, it is no longer possible as of the EIP 150 hardfork.

F. SECURIFY

Securify [20], [21] is a security analysis tool for Ethereum smart contracts, which currently supports more than 37 vulnerabilities including reentrancy, locked Ether, transaction order dependence, and unrestricted write. Together with an input contract, it takes as input a set of security patterns written in a specialized domain-specific language. More specifically, a security property is encoded into a set of compliance and violation patterns, each of which ensures that a contract satisfies and violates the given property, respectively. Such patterns are checked using the Soufflé Datalog solver [34] against the semantic facts obtained from the contract by applying static analysis such as data- and control-flow analysis. In contrast to symbolic execution-based tools such as Mythril [16] and Oyente [18], which do not guarantee to explore every program path, Securify analyzes every contract behavior, thus avoiding false negatives. Securify aims to guarantee that if a contract matches a compliance (resp. violation) pattern, then it definitely complies with (resp. violates) the corresponding security property. However, as discussed in [35], most of the security patterns proposed in [20] are not sound and can produce both false positives and false negatives.

G. SLITHER

Slither [22], [23] is an open-source Solidity static analysis framework written in Python 3, which supports automated detection of about 45 vulnerabilities and code optimizations that the compiler misses, and visualization of the information about contract details, enhancing developers' code comprehension. Given a Solidity contract source code, Slither takes as input its abstract syntax tree generated by the Solidity compiler, and recovers its inheritance graph, control flow graph, and list of expressions. Then, Slither transforms the contract code into an intermediate representation called SlithIR, which uses static single assignment form [36] to facilitate the analysis, and applies the usual program analysis techniques such as data-flow analysis and taint tracking. The authors of [22] compares Slither with other static analysis tools such as Securify [20], SmartCheck [24], and Solhint [26] with respect to their capability to detect reentrancy vulnerabilities using 1,000 contracts obtained from Etherscan (https://etherscan.io/), and show that Slither outperforms the other tools for detecting reentrancy vulnerabilities with respect to performance, robustness, and accuracy.

H. SMARTCHECK

SmartCheck [24], [25] is an efficient static analysis tool for Ethereum smart contracts to detect security vulnerabilities and other code issues. It uses an XML-based intermediate representation (IR) to which Solidity source code is translated. Potential vulnerabilities are then detected by applying XPath [37] patterns on the generated IR. Although SmartCheck is very fast when compared with other analysis tools [10], since it only performs relatively simple lexical and syntactic analysis, it cannot detect some severe bugs requiring more advanced techniques such as taint analysis. It has also shown that SmartCheck produces a large number of false positives in the experiment on the reentrancy vulnerability detection using 1,000 contracts [22]. An online version of SmartCheck with more security patterns than the GitHub version is available at https://tool.smartdec.net/.

I. SOLHINT

Solhint [26] is an open-source linter for Solidity smart contracts, similar to Ethlint [13]. It can be used not only to validate if the Solidity code complies with the style guide and best coding practices but also to detect syntax-related security vulnerabilities. In addition, the user can customize the predefined rule sets and add new rules if necessary. Solhint has shown to be fast and robust, but produce a large number of false positives in the experiment on the reentrancy vulnerability detection [22].

J. SOL-PROFILER

Sol-profiler [27] is a CLI tool to help the user to visualize and review Solidity smart contracts by listing down various properties of every contract method. More specifically, for each method, it specifies the contract, interface, or library to which it belongs, its name and parameter types, its visibility (external, public, internal, or private), if it is a view or pure function, its return type, and its modifiers. Therefore, by using Sol-profiler, the user can easily identify the properties of the contract methods and check if there are errors. However, Sol-profiler does not guarantee any security properties of smart contracts.

K. VERISMART

VeriSmart [28], [29] is a highly precise verification tool for detecting arithmetic bugs such as an integer overflow and underflow in Ethereum smart contracts. It automatically discovers the transactions invariants of smart contracts, which enable to analyze them effectively and exhaustively. More precisely, it iteratively generates candidate transaction invariants and validates them using an off-the-shelf SMT solver as in the usual counter example-guided inductive synthesis (CEGIS) framework [38]. By experimentally comparing VeriSmart with other analysis tools that can detect arithmetic bugs such as Manticore [14], Mythril [16], Osiris [33], and Oyente [18], using 60 contracts that contains arithmetic vulnerabilities [39], the authors of [28] show that VeriSmart far outperforms the abovementioned analyzers and detects all arithmetic bugs with a negligible false positive rate. Since VeriSmart outperforms Osiris, which can detect only integer-related bugs, we do not include the latter in our proposed evaluation tool.

IV. THE PROPOSED EVALUATION TOOL

A. DESIGN OF THE PROPOSED TOOL

A number countermeasures have been introduced to detect vulnerabilities in smart contract applications, as mentioned in the previous section, but their effectiveness has not been studied well. It is even not clear which countermeasures are most effective for each type of vulnerability. When new countermeasures are proposed, it is definitely necessary to conduct comparative performance evaluation with existing ones. However, since different countermeasures could require different environments for installation and execution (e.g., different versions of the Solidity compiler and Z3 theorem prover, etc.) and their verification outputs are produced in different formats, it is not a simple task to perform a comparative study of countermeasures and compare the analysis results. In particular, when new datasets are available, one needs to re-execute all available countermeasures, preprocess their verification outputs, and analyze the result under various performance measures. To avoid such time-consuming tasks, we provide a software tool that can

- easily be extended with existing/new countermeasures and labeled smart contract codes,
- facilitate comparison of the countermeasures by automatically analyzing their verification outputs in terms of various performance measures and arranging the results in tables and graphs, and thus
- help the user to identify the most effective countermeasures for each vulnerability.

Fig. 1 shows the overall structure of our proposed evaluation tool. In the proposed tool, each countermeasure is offered in the form of a Docker image using OS-level virtualization and operates within the Docker container, making it easy to meet all operational requirements. This approach helps to effectively manage the use of computational resources (CPU, memory) in the system, since each countermeasure is containerized only while actually analyzing the target code. A set of different versions of the Solidity compiler is provided as a single Docker image and can be used in different containers where countermeasures operate. This design is effective because it eliminates the need to update all Docker images of the existing countermeasures when a new version of the compiler is required to convert a new target code into binary code.

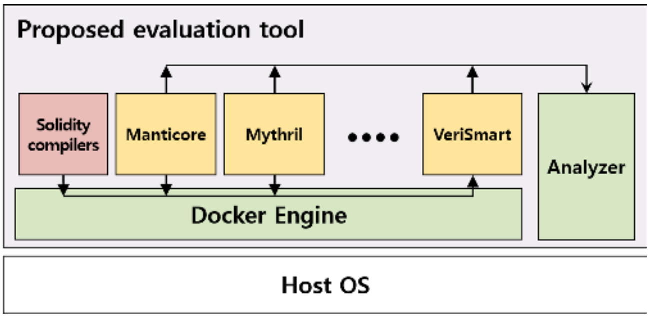


FIGURE 1: Overall structure of the proposed evaluation tool.

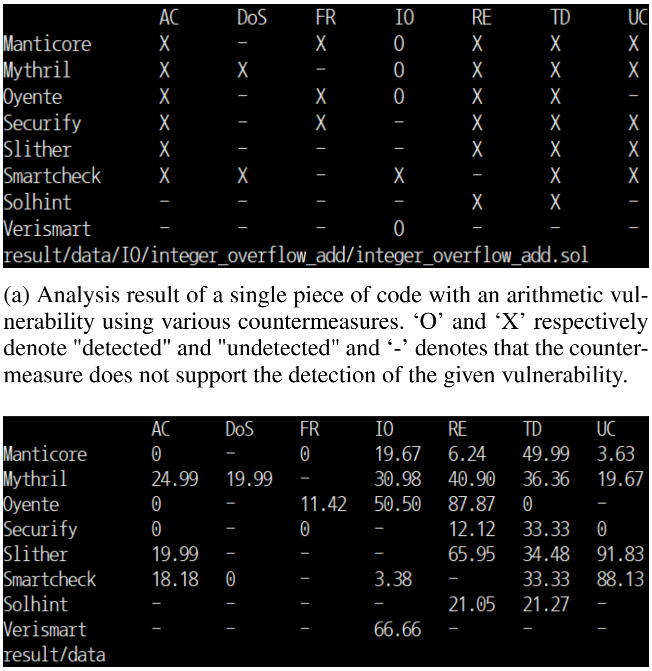
The analyzer module analyzes the verification outputs generated by each countermeasure and demonstrates the verification performance using various statistical indicators. More precisely, it preprocesses the verification outputs of each countermeasure to check if some vulnerabilities were found in each code in the benchmark dataset. Then, for each countermeasure and its verification outputs, the analyzer module automatically computes various performance measures such as the numbers of true positives, false positives, true negatives, and false negatives, precision, recall, accuracy, F1-score, and the area under the curve (AUC), as shown in Table 2. Finally, it organizes the analysis results and presents them using tables and graphs as shown in Fig. 2, 3, and 4, making it easy to conduct comparative studies for each type of vulnerability. In particular, since the verification performance is represented using various performance measures, users can identify the most effective countermeasure according to their own interests. (Here we note that, different users can place higher importance on different measures. For example, some users may give higher priority to countermeasures that maximize the number of true positives than those that have the minimum number of false positives, while others may prefer the opposite.) This feature can be useful if the proposed tool is used to verify smart contract codes in practice. In addition to selective application of each countermeasure, an effective subset of countermeasures can automatically be selected/recommended depending on the target vulnerabilities, user interests, and constraints.

B. COMPARATIVE STUDY USING THE PROPOSED TOOL

Using the proposed tool, we evaluated the performance of the state-of-the-art countermeasures with 237 pieces of la-

TABLE 2: Performance measures used in the proposed tool. The value of precision, recall, accuracy, and F1-score ranges from 0 to 100, and that of AUC ranges from 0 to 1.

Measure	Description
TP	Number of true positives, i.e., contracts having vulnerability are correctly identified.
FP	Number of false positives, i.e., contracts without vulnerability are determined vulnerable.
TN	Number of true negatives, i.e., contracts without vulnerability are identified non-vulnerable.
FN	Number of false negatives, i.e., contracts having vulnerability are determined non-vulnerable.
Precision ($\mathcal{P}$)	Percentage of true positives among those determined as positive, i.e., $\mathcal{P}$ = TP / (TP + FP).
Recall ($\mathcal{R}$)	Percentage of true positives among the actual positives, i.e., $\mathcal{R}$ = TP / (TP + FN).
Accuracy	Percentage of correctly identified cases among all cases, i.e., Accuracy = (TP + TN) / (TP + TN + FP + FN).
F1-score	Harmonic mean of precision and recall, i.e., F1-score = $2\mathcal{PR}/(\mathcal{P}+\mathcal{R})$ = 2TP / (2TP + FP+FN).
AUC	Area under a receiver operating characteristic (ROC) curve created by plotting the true positive rate (i.e., recall, sensitivity) against the false positive rate (i.e., fall-out, 1 - specificity) at various thresholds.



```
           AC    DoS   FR    IO    RE    TD    UC
Manticore  X     -     X     O     X     X     X
Mythril    X     X     -     O     X     X     X
Oyente     X     -     X     O     X     X     -
Securify   X     -     X     -     X     X     X
Slither    X     -     -     -     X     X     X
Smartcheck X     X     -     X     -     X     X
Solhint    -     -     -     -     X     X     -
Verismart  -     -     -     O     -     -     -
result/data/IO/integer_overflow_add/integer_overflow_add.sol
```

(a) Analysis result of a single piece of code with an arithmetic vulnerability using various countermeasures. ‘O’ and ‘X’ respectively denote "detected" and "undetected" and ‘-’ denotes that the countermeasure does not support the detection of the given vulnerability.

```
           AC     DoS    FR     IO     RE     TD     UC
Manticore  0      -      0      19.67  6.24   49.99  3.63
Mythril    24.99  19.99  -      30.98  40.90  36.36  19.67
Oyente     0      -      11.42  50.50  87.87  0      -
Securify   0      -      0      -      12.12  33.33  0
Slither    19.99  -      -      -      65.95  34.48  91.83
Smartcheck 18.18  0      -      3.38   -      33.33  88.13
Solhint    -      -      -      -      21.05  21.27  -
Verismart  -      -      -      66.66  -      -      -
result/data
```

(b) F1-score of each countermeasure for the benchmark dataset described in Table 3.

FIGURE 2: Example results of using the proposed tool.

beled code collected from the SWC registry (Smart Contract Weakness Classification and Test Cases) [40], SmartBugs SB curated dataset [41], VeriSmart-benchmarks [39], Zeus dataset [42], and eThor dataset [43]. Each code either has a single type of vulnerability or is known to be secure, i.e., without any vulnerability. The number of pieces of code for each vulnerability is arranged in Table 3. The proposed tool arranges the evaluation results in a unified manner as shown in Fig. 2 and produces graphs for each measure as shown in Fig. 3 and 4, which allows an easy comparative study and cross-validation among the countermeasures.

TABLE 3: The number of smart contracts for testing each countermeasure for each type of vulnerability. Secure denotes smart contracts having no vulnerability.

AC	DoS	FR	IO	RE	TD	UC	Secure	TOTAL
18	6	4	55	31	5	52	66	237

Tables 3–8 respectively shows the TP, FP, precision, recall, accuracy, and F1-score of each countermeasure for each type of vulnerability for the dataset described in Table 3. We omit the TN and FN as they are easily obtained from the the FP and TP, respectively. In the tables, ‘-’ means that the countermeasure does not support the detection of the corresponding vulnerability. In Table 3, `TOTAL` represents the number of smart contracts having the corresponding vulnerability, whereas in Table 4, it represents the number of those not having the corresponding vulnerability. We additionally show the F1-score of each countermeasure for each type of vulnerability in Fig. 3, which takes both precision and recall into consideration and thus is a more appropriate metric for imbalanced datasets. As our dataset is highly imbalanced, the F1-score is much lower than the accuracy for every case, but it is much more useful than the accuracy for comparing the performance of various countermeasures.

Overall, for AC, every countermeasure shows a low detection rate of vulnerable code. That is, for all countermeasures, more than 80% of vulnerable codes are not detected. Mythril detects three of the 18 vulnerable codes, showing the largest TP value. However, considering the FP value together, Slither, which represents a slightly smaller TP value, can be more effective since it shows 100% precision and better accuracy.

Only Mythril and Oyente work for DoS and FR, respectively. One out of six vulnerable codes with DoS is detected by Mythril, and two out of four vulnerable codes with FR are detected by Oyente. Both countermeasures produce more false positives (compared to the true positives), resulting in small values in both precision and recall. These results can be interpreted that neither countermeasure makes a sufficiently meaningful contribution to DoS and FR detection.

VeriSmart successfully detects all vulnerable codes with IO and reports a 100% recall value. However, it also generates 55 false positives, resulting in the precision of 50%. In general, a large number of false positives require additional manual examinations, which can be a large overhead. In that sense, Manticore that detects six out of 55 vulnerable codes without generating any false positives (100 % precision) may be preferred.

Oyente may be the most effective tool for detecting RE, as shown by the highest value in F1-score (Fig. 3) and good performance for all measures (refer to Tables 5, 6 and 7). More specifically, 29 out of 31 vulnerable codes are detected

TABLE 3: The number of true positives (TP).

TOOL	AC	DoS	FR	IO	RE	TD	UC
Manticore	0	-	0	6	1	2	1
Mythril	3	1	-	11	9	2	6
Oyente	0	-	2	50	29	0	-
Securify	0	-	0	-	2	1	0
Slither	2	-	-	-	31	5	45
SmartCheck	2	0	-	1	-	1	52
Solhint	-	-	-	-	6	5	-
VeriSmart	-	-	-	55	-	-	-
TOTAL	18	6	4	55	31	5	52

TABLE 4: The number of false positives (FP).

TOOL	AC	DoS	FR	IO	RE	TD	UC
Manticore	0	-	0	0	0	1	2
Mythril	3	3	-	5	4	4	3
Oyente	0	-	29	93	6	10	-
Securify	2	-	4	-	0	0	0
Slither	0	-	-	-	32	19	1
SmartCheck	2	11	-	3	-	0	14
Solhint	-	-	-	-	20	37	-
VeriSmart	-	-	-	55	-	-	-
TOTAL	219	231	233	182	206	232	185

TABLE 5: Precision (%).

TOOL	AC	DoS	FR	IO	RE	TD	UC
Manticore	0	-	0	100	100	66.7	33.3
Mythril	50	25	-	68.8	69.2	33.3	66.7
Oyente	0	-	6.5	35	82.9	0	-
Securify	0	-	0	-	100	100	0
Slither	100	-	-	-	49.2	20.8	97.8
SmartCheck	50	0	-	25	-	100	78.8
Solhint	-	-	-	-	23.1	11.9	-
VeriSmart	-	-	-	50	-	-	-

TABLE 6: Recall (%).

TOOL	AC	DoS	FR	IO	RE	TD	UC
Manticore	0	-	0	10.9	3.2	40	1.9
Mythril	16.7	16.7	-	20	29	40	11.5
Oyente	0	-	50	90.9	93.5	0	-
Securify	0	-	0	-	6.5	20	0
Slither	11.1	-	-	-	100	100	86.5
SmartCheck	11.1	0	-	1.8	-	20	100
Solhint	-	-	-	-	19.4	100	-
VeriSmart	-	-	-	100	-	-	-

TABLE 7: Accuracy (%).

TOOL	AC	DoS	FR	IO	RE	TD	UC
Manticore	97	-	99.3	91.8	95	99.3	91.1
Mythril	97	98.7	-	91.8	95.6	98.8	91.8
Oyente	97	-	94.8	83.5	98.7	97.5	-
Securify	96.6	-	98.7	-	95.1	99.3	91.3
Slither	97.3	-	-	-	94.6	96.8	98.7
SmartCheck	97	97.1	-	90.4	-	99.3	97.6
Solhint	-	-	-	-	92.4	93.8	-
VeriSmart	-	-	-	90.8	-	-	-

TABLE 8: F1-score (%).

TOOL	AC	DoS	FR	IO	RE	TD	UC
Manticore	0	-	0	19.7	6.2	50	3.6
Mythril	25	20	-	31	40.9	36.4	19.7
Oyente	0	-	11.4	50.5	87.9	0	-
Securify	0	-	0	-	12.1	33.3	0
Slither	20	-	-	-	66	34.5	91.8
SmartCheck	18.2	0	-	3.4	-	33.3	88.1
Solhint	-	-	-	-	21.1	21.3	-
VeriSmart	-	-	-	66.7	-	-	-

while only six false positives are produced (among 206 secure codes). Slither may be considered competitive in that it completely detects every vulnerable code (even though it produces 32 false positives, showing 49.2% precision and 100% recall).

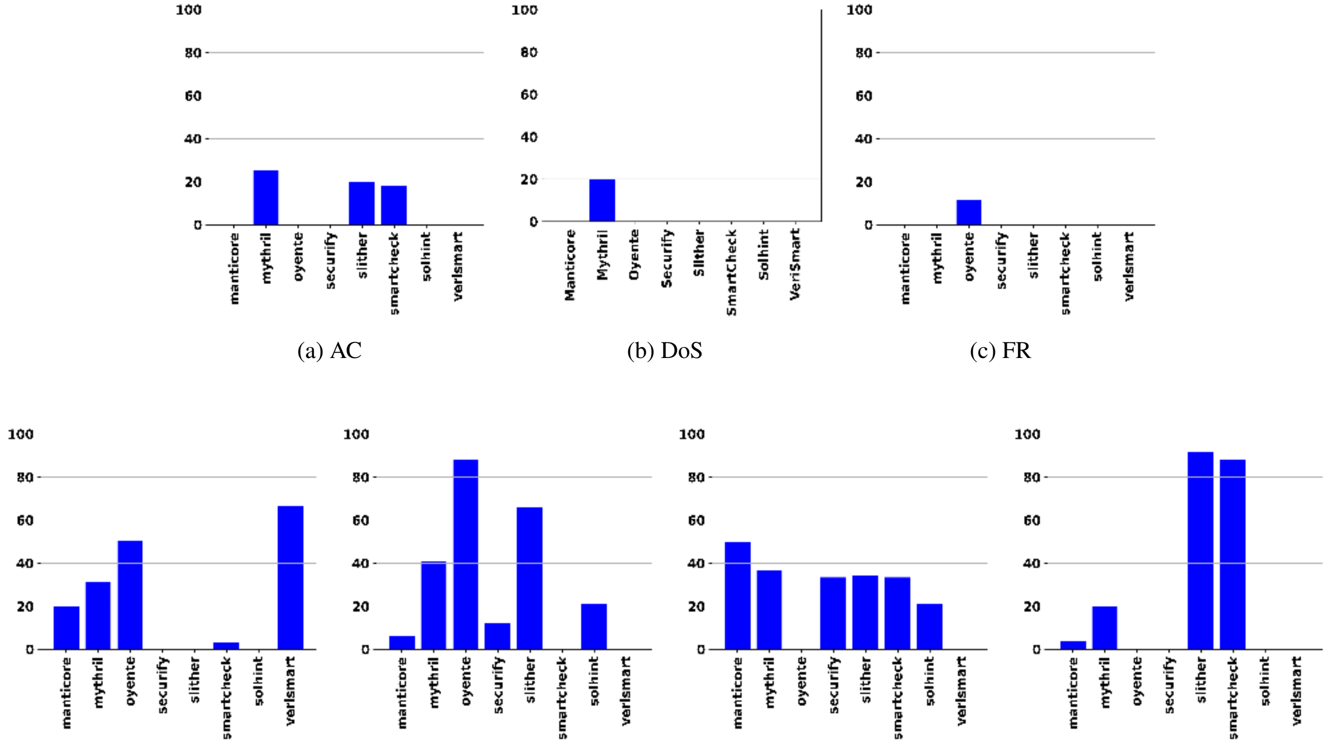
It is confirmed that both Slither and Solhint completely detect five vulnerable codes having TD. However, Slither can be recommended more preferentially since it produces about half of the false positives in Solhint. As for UC, it is reported that all vulnerable codes are detected by SmartCheck which produces 14 false positives, resulting in the precision of 78.8% and F1-score of 88.1%. Slither is also effective in detecting UC. It detects 45 out of 52 vulnerable codes while generating only one false positive, showing good performance for all measures.

In Fig. 4, we also show the performance of the countermeasures using their ROC curve and AUC value. Since our dataset is imbalanced, the AUC is also an important measure to be considered. Note that the AUC values in Fig. 4 do not necessarily coincide with the F1-scores in Fig. 3. Following the general guidelines in [44], we consider the countermeasure to be acceptable, excellent, and outstanding if its AUC value is greater than or equal to 0.7, 0.8, and 0.9, respectively. In this regard, there is no effective countermeasure for AC, DoS, and FR; Oyente is excellent and VeriSmart is outstanding at identifying IO; Oyente and Slither are outstanding for RE; Slither and Solhint are outstanding for TD; and finally Slither and SmartCheck are outstanding for UC.

The evaluation results can be summarized as follows:

- In general, countermeasures that identify many vulnerable codes also tend to be less precise and accurate since they also produce much more false positives.
- There is no effective countermeasure for detecting the ‘Access Control’, ’Denial of Service’, and ‘Front-Running’ vulnerabilities yet.
- Vulnerable codes with ‘Integer Overflow’ and ‘Timestamp Dependence’ are completely detected. However, the performance of the countermeasures need to be further improved in order to reduce false positives.
- As for the ‘Reentrancy’ and ‘Unchecked Low Level Call’ vulnerabilities, effective countermeasures that show both high precision and high recall values are identified.

V. DISCUSSION



(a) AC

(b) DoS

(c) FR

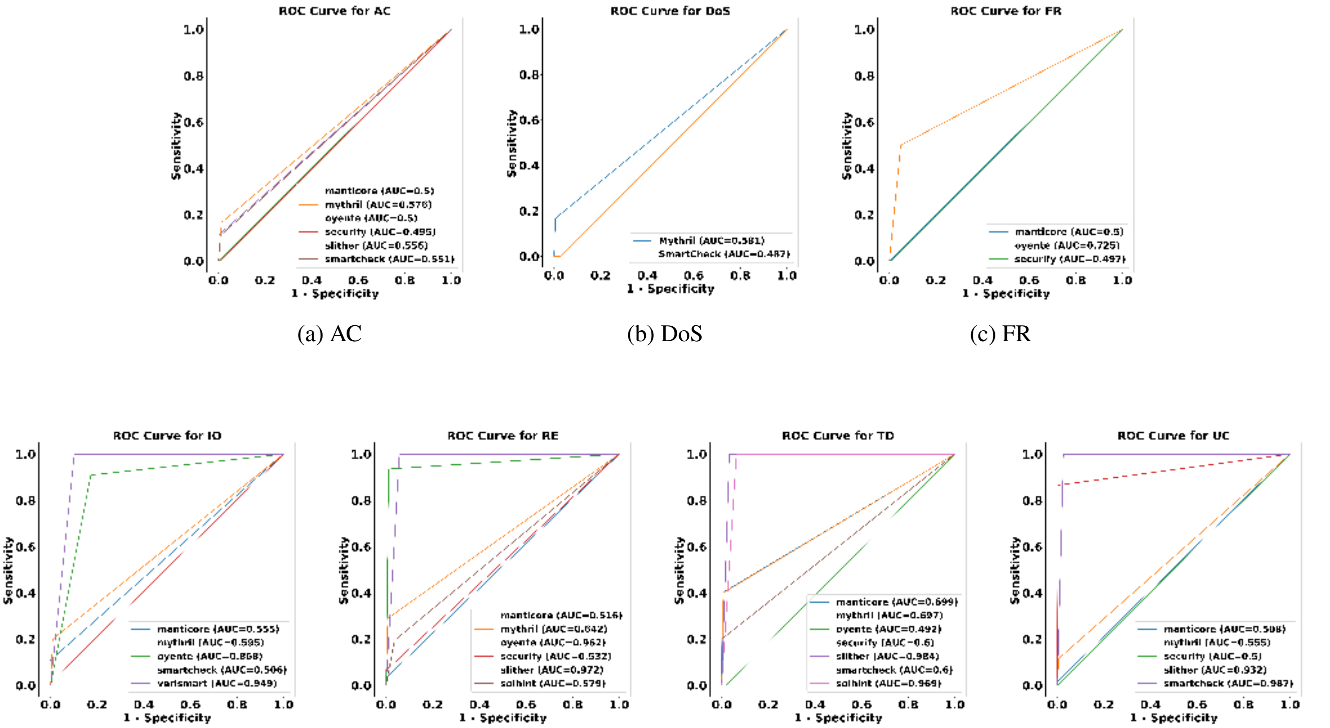
(d) IO

(e) RE

(f) TD

(g) UC

FIGURE 3: F1-score of each countermeasure for each vulnerability.



(a) AC

(b) DoS

(c) FR

(d) IO

(e) RE

(f) TD

(g) UC

FIGURE 4: ROC curve and AUC of each countermeasure for each vulnerability.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2021.3091317, IEEE Access

A. RESULTS FROM OUR COMPARATIVE STUDY

Similar to ours, previous studies such as [9], [10], [22], [28] have also empirically compared various countermeasures using real-world smart contracts and discussed their performance. In [9], four countermeasures such as Mythril, Securify, SmartCheck, and Oyente were evaluated using 10 representative smart contracts, and the results suggested that SmartCheck was statistically most effective in terms of accuracy and ROC, while Mythril had the least number of false positives. This result is consistent with our evaluation results in the case of UC. However, due to a limited number of test codes, the effectiveness of Oyente against IO did not seem to be well understood. The authors of [22] proposed Slither and conducted performance comparisons with Securify, SmartCheck, and Solhint in RE detection using two famous contracts (DAO and SpankChain), which are vulnerable to RE, and 1,000 unlabeled contract data. They reported that Slither overwhelmed the other three countermeasures in terms of accuracy, execution time, and robustness, which is the same as in our evaluation results. The performance of Oyente, which was not covered in their work, is newly verified in our work, and it is discussed that Oyente could be more effective than Slither in that it generates a much smaller number of false positives for RE detection. In [28], the authors introduced VeriSmart, a new method for IO detection, and compared its performance with that of Manticore, Mythril, Osiris, and Oyente using 60 labeled vulnerable contracts. Their evaluation showed that VeriSmart successfully detected all vulnerable codes with a negligible false positive rate (0.41%). In our study using an increased number of codes, VeriSmart's effectiveness could also be confirmed. However, in our study, it incurred a much higher false positive rate. In [10], comprehensive evaluation for the nine representative countermeasures were performed using a dataset of 69 labeled vulnerable contracts and 47,518 unlabeled contracts. The authors reported that Mythril was the most accurate countermeasure, showing 27% accuracy, when considered the vulnerabilities altogether. However, this report is slightly different from our findings. In our work, which used more labeled codes and measured various evaluation metrics for each vulnerability, Mythril is not recommended due to its low value of precision and recall.

Obviously, our experimental results are partially consistent and complementary with those from previous studies mentioned above. Here, we note that in our work, more countermeasures are evaluated with much more labeled codes, and their performance is shown with various measures. Hence, we believe that our comparative study could provide more reliable insights into the state-of-the-art countermeasures and help developers choose countermeasures that better suit their purpose under given conditions.

B. THREATS TO VALIDITY

The limitations of our evaluation are summarized as follows. In this work, we collected more labeled data than previous studies to derive more reliable evaluation results. However, as in other related work, a few smart contracts may have incorrect labels, as it is very challenging to manually examine the code. Moreover, our dataset is imbalanced in that the number of safe smart contracts is much larger than the number of vulnerable smart contracts. In particular, among the 237 smart contracts, only six, four, and five vulnerable contracts for DoS, FR and TD are included, respectively. Hence, in most cases, the detection accuracy of countermeasures against each vulnerability is highly reported. Since new datasets and countermeasures can be easily added to our tool, however, we believe that our tool can contribute to achieving more accurate evaluation results with more data in the future.

VI. CONCLUSIONS

In this paper, we revisited smart contracts using the Ethereum blockchain technology and summarized various vulnerability issues of smart contract applications. A number of countermeasures were briefly introduced and discussed. To assess the effectiveness of the countermeasures, we designed and implemented a software tool that facilitates comparative evaluations of the countermeasures. Using the tool and 237 labeled benchmark codes, we evaluated state-of-the-art vulnerability detection schemes. The evaluation results indicate that countermeasures that exhibit a larger TP value often generate a much larger number of false positives, resulting in very low precision and accuracy. In addition, among the state-of-the-art countermeasures, Oyente and Slither are most effective for RE detection; Slither could be recommended for detection of TD and UC; and VeriSmart could be recommended for IO detection. Using our tool, researchers can easily conduct performance comparisons between their own countermeasure and other state-of-the-art schemes with a variety of performance metrics. As for practitioners, they can exploit our tool to find various vulnerabilities within their smart contract applications. Since in our tool, smart contracts can be examined by a number of countermeasures simultaneously, vulnerabilities can be easily identified. We believe that our proposed tool will be effective in a future verification study of smart contracts and will contribute to the development of practical and secure smart contract applications.

APPENDIX. USAGE OF THE PROPOSED TOOL

The proposed software tool is open to public via the website https://github.com/93suhwan/uscv. This section details how to use it.

A. INSTALLATION

As mentioned in Section IV-A, each countermeasure is included in the proposed software tool in the form of a Docker image. To create a Docker image, a dockerfile can be run under the following command:

```
$ docker build [dockerfile]
```

arguments	
[dockerfile]	location of the dockerfile

The content of a dockerfile is as follows:

```
From ubuntu:18.04
RUN [installCommand]
ENTRYPOINT [exeCommand]

/*
- 1st line indicates a layer is created
  from the ubuntu:18.04 Docker image
- 2nd line is for building a countermeasure
  by executing [installCommand]
- 3rd line specifies the execution command
  ([exeCommand]) for each countermeasure
  (which would run by default)
*/
```

Using the Docker image, a Docker container is generated and executed under the following command:

```
$ docker build -t [dockerImage] [dockerfile]
```

arguments	
[dockerImage]	name of the Docker image of a countermeasure
[dockerfile]	location of the dockerfile

Since multiple versions of the Solidity compiler may be required during the process of compiling the source files, the proposed tool has a Docker image (.solc) that includes different versions of the Solidity compiler. The installed compilers can be used at each container under the following command:

```
$ docker run -v [curDir]:[containerDir] \
                [dockerImage] [options]
```

arguments	
[curDir]	current directory in a host (e.g., $ (pwd)/.solc)
[containerDir]	directory in a container (e.g., /root/.solc)
[dockerImage]	Docker image to be used for testing
[options]	built-in options of the Docker container (countermeasure)

All of these processes for the 11 countermeasures discussed in Section III are executed automatically by running “createContainers.sh”.

B. EXECUTION

The file named “testing.sh" is used to execute each countermeasure. It preprocesses the source file and determines the version of the compiler that can be used for compiling. Then, it can be run with the generalized options as follows:

```
$ testing.sh -t [schemeName] \
             -f [srcFile] -l [timeout]
(e.g.,)
>> testing.sh -t oyente -f test.sol \
              -o "-dl 10 -r" -l 100
```

options	usage
-t	used to specify the countermeasure’s name [schemeName]
-f	used to specify the target code to be tested [srcFile]
-l	used to specify a timeout value (If a timer expires with the given timeout, the program is forced to quit.)
-o	used to specify the options that each countermeasure uniquely supports

The file named “execution.sh” is provided to run multiple countermeasures. “execution.sh” supports the following options:

```
$ execution.sh -f/d [srcFile/dirName] \
               -t [toolName]
(e.g.,)
>> execution.sh -d ./curDir -v AC -l 100
```

options	usage
-f	used to specify the name of a target code [srcFile] (for testing a single target code)
-d	used to specify the name of a directory [dirName] (for testing multiple codes within the directory)
-t	used to specify a countermeasure [schemeName] [-t **Aux**] : checking grammar and predefined properties applying Echidna, Ethlint, and Sol-profiler [-t **Vul**] : checking vulnerabilities by applying eight countermeasures but the three mentioned above [-t **All**] : applying all countermeasures
-v	used to specify a type of vulnerability - AC (Access control), DoS (Denial of service), FR (Front-running), IO (Integer overflow/underflow), RE (Reentrancy), TD (Timestamp dependency), UC (Unchecked low level call) - Countermeasures associated with the given vulnerability are selectively executed.
-l	used to specify a timeout value (When a time expires, countermeasures are forced to quit.)

The analysis results are recorded in the file named `tool_name.txt` under the directory of “./result”.

C. ADDING NEW COUNTERMEASURES AND DATA

When new countermeasures are proposed, they can easily be integrated into our proposed tool and evaluated with the embedded benchmark data. To this end, the file named “addScheme.sh” is provided. By specifying meta-information on a new countermeasure as arguments, the new scheme can simply be included into the system.

```
$ addScheme.sh -l/n [dirName/imageName] \
              -e [cmd] -o [option] \
              -M [word]
(e.g.,)
>> addScheme.sh -l dockerfiles/smartcheck \
      -e smartcheck -o p \
      -a SOLIDITY_TX_ORIGIN \
      -d SOLIDITY_OVERPOWERED_ROLE \
      -i SOLIDITY_VAR|SOLIDITY_UINT_CANT \
      -t SOLIDITY_EXACT_TIME \
      -u SOLIDITY_UNCHECKED_CALL
```

arguments	
-l	directory having a dockerfile of the countermeasure [dirName]
-n	name of a Docker image created by the user [imageName]
-e	execution command of the countermeasure [cmd]
-o	[option] when the countermeasure requires a flag to input the target code
-M	unique word [word] in the result strings generated by the countermeasure against detected vulnerabilities. values of M : a (AC), d (DoS), f (FR), i (IO), r (RE), t (TD), u (UC)

The installed countermeasures can also be removed from the tool as follows.

```
$ removeScheme.sh [countermeasureName]
(e.g.,)
>> removeScheme.sh mythril
```

When labeled codes are newly collected, they can be added to our tool and used for the analysis.

```
$ addData.sh -d/f [dirName/fileName] \
             -c [vulType]
(e.g.,)
>> addData.sh -f example.sol -c AC
```

arguments	
-d	directory that has the target codes [dirName]
-f	name of the target code [fileName]
-c	vulnerability type [vulType] (AC, DoS, FR, IO, RE, TD, UC)

REFERENCES

[1] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Tech. Rep., 2008.

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, pp. 1–32, 2014.

[3] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on Ethereum smart contracts (SoK)," in *Principles of Security and Trust*, M. Maffei and M. Ryan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186.

[4] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Future Generation Computer Systems*, vol. 107, pp. 841–853, 2020.

[5] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections," *IEEE Access*, vol. 8, pp. 24 416–24 427, 2020.

[6] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A survey on Ethereum systems security: Vulnerabilities, attacks, and defenses," *ACM Comput. Surv.*, vol. 53, no. 3, Jun 2020.

[7] NCC Group, "Decentralized application security project (or DASP) top 10," Available online: https://dasp.co/ (accessed on 25 March 2021).

[8] M. di Angelo and G. Salzer, "A survey of tools for analyzing Ethereum smart contracts," in *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*, 2019, pp. 69–78.

[9] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical vulnerability analysis of automated smart contracts security testing on blockchains," in *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering*, ser. CASCON '18. USA: IBM Corp., 2018, pp. 103–113.

[10] T. Durieux, J. a. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 530–541.

[11] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: Effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 557–560.

[12] Trail of Bits, "Echidna: A fast smart contract fuzzer," Available online: https://github.com/crytic/echidna (accessed on 25 March 2021).

[13] "Ethlint," Available online: https://github.com/duaraghav8/Ethlint (accessed on 25 March 2021).

[14] 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)*, 2019, pp. 1186–1189.

[15] Trail of Bits, "Manticore," Available online: https://github.com/trailofbits/manticore/ (accessed on 25 March 2021).

[16] B. Mueller, "Smashing Ethereum smart contracts for fun and real profit," in *The 9th Annual HITB Security Conference*, 2018, available online: https://github.com/b-mueller/smashing-smart-contracts (accessed on 25 March 2021).

[17] ConsenSys, "Mythril," Available online: https://github.com/ConsenSys/mythril (accessed on 25 March 2021).

[18] 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*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, pp. 254–269.

[19] "Oyente," Available online: https://github.com/melonproject/oyente (accessed on 25 March 2021).

[20] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 67–82.

[21] "Securify 2.0," Available online: https://github.com/eth-sri/securify2 (accessed on 25 March 2021).

[22] J. Feist, G. Greico, and A. Groce, "Slither: A static analysis framework for smart contracts," in *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, ser. WETSEB '19. IEEE Press, 2019, pp. 8–15.

[23] Trail of Bits, "Slither, the Solidity source analyzer," Available online: https://github.com/crytic/slither (accessed on 25 March 2021).

[24] 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*, ser. WETSEB '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 9–16.

[25] "SmartCheck," Available online: https://github.com/smartdec/smartcheck (accessed on 25 March 2021).

[26] "Solhint," Available online: https://protofire.github.io/solhint/ (accessed on 25 March 2021).

[27] "Sol-profiler," Available online: https://github.com/Aniket-Engg/sol-profiler (accessed on 25 March 2021).

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

[29] "VeriSmart," Available online: https://github.com/kupl/VeriSmart-public (accessed on 25 March 2021).

[30] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of Haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, pp. 268–279.

[31] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in Ethereum smart contracts," in *Proceedings of the 28th USENIX Conference on Security Symposium*, ser. SEC'19. USA: USENIX Association, 2019, p. 1591–1607.

[32] 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*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 653–663.

[33] 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*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 664–676.

[34] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds. Cham: Springer International Publishing, 2016, pp. 422–430.

[35] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "eThor: Practical and provably sound static analysis of Ethereum smart contracts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 621–640.

[36] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '88. New York, NY, USA: Association for Computing Machinery, 1988, pp. 12–27.

[37] "XML Path Language (XPath) 2.0," Available online: https://www.w3.org/TR/xpath20/ (accessed on 25 March 2021).

[38] A. Solar-Lezama, "Program sketching," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 5–6, pp. 475–495, October 2013.

[39] "VeriSmart-benchmarks," Available online: https://github.com/kupl/VeriSmart-benchmarks (accessed on 25 March 2021).

[40] "SWC registry (smart contract weakness classification and test cases)," Available online: https://swcregistry.io/ (accessed on 25 March 2021).

[41] "SmartBugs: A framework to analyze Solidity smart contracts," Available online: https://github.com/smartbugs/smartbugs (accessed on 25 March 2021).

[42] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[43] "eThor: Sound static analysis for Ethereum smart contracts," Available online: https://secpriv.wien/ethor/ (accessed on 25 March 2021).

[44] D. W. Hosmer, S. Lemeshow, and R. X. Sturdivant, *Applied Logistic Regression*, 3rd ed. Wiley, 2013.

SUHWAN JI received the B.S. (2017) and M.S. (2019) degrees in computer science from Kangwon National University, Korea. He is currently a Ph.D. student in the Interdisciplinary Graduate Program in Medical Bigdata Convergence, majoring in AI and software, at Kangwon National University. His research interests include programming languages, machine learning, and blockchain.

DOHYUNG KIM received the BS degree in information and computer engineering from Ajou University, Suwon, South Korea, in February 2004, and the PhD degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in August 2014. He was a postdoc researcher and research professor with the Department of Computer Engineering, Sungkyunkwan University from 2014 to 2017. In 2018, he was an assistant professor with the Department of Software and Computer Engineering, Ajou University. Currently, he is an assistant professor with the Department of Computer Science and Engineering, Kangwon National University. His research interests include the design and analysis of computer networking and wireless communication systems especially for future Internet architectures.

HYEONSEUNG IM received the B.S. degree in computer science from Yonsei University, Korea, in 2006 and the Ph.D. degree in computer science and engineering from Pohang University of Science and Technology (POSTECH), Korea, in 2012. From 2012 to 2015, he was a Postdoctoral Researcher with the Laboratory for Computer Science at Université Paris-Sud and with the Tyrex team, Inria, France. He is currently an Associate Professor with the Department of Computer Science and Engineering, Kangwon National University, Korea. His research interests include programming languages, logic in computer science, big data analysis and management, machine learning, precision medicine, blockchain, and information security.

• • •