A modular testing approach for detecting smart contract vulnerabilities

Mohammad Reza Talebi
MDSE Research Group, Department of Software Engineering
University of Isfahan
Isfahan, Iran
talebi@eng.ui.ac.ir

Shekoufeh Kolahdouz Rahimi
MDSE Research Group, Department of Software Engineering
University of Isfahan
Isfahan, Iran
sh.rahimi@eng.ui.ac.ir

Behrouz Tork Ladani
MDSE Research Group, Department of Software Engineering
University of Isfahan
Isfahan, Iran
ladani@eng.ui.ac.ir

***Abstract*— Due to blockchain transparency and immutability, transactions and smart contracts codes are disclosed to anyone, and smart contracts will be immutable after development. Accordingly, various vulnerabilities have led to the destruction of smart contracts. There are much works on static and dynamic detection. Many research leads static and dynamic detection to automatic detection, while automatic detection does not provide modular testing and does not consider smart contracts business logic. Also, smart contracts interact with each other, and an included vulnerability causes the related contracts to be vulnerable. This paper proposes a modular testing approach that detects three classes of vulnerabilities over the business logic and proposes three unit test patterns that detect reentrancy, phishable tx.origin, and missing protection against signature replay vulnerabilities. By using unit test patterns within the Hardhat environment, developers detect mentioned vulnerabilities. To evaluate, we collect a set of vulnerable and non-vulnerable Ethereum smart contracts as a new benchmark, apply the proposed unit test patterns, and compare the results with two updated symbolic execution tools. The results show the applicability of the proposed modular approach for detecting the vulnerability of single and twice smart contracts. By using the proposed approach, developers can test the smart contract implementation.**

Keywords—modular testing, smart contract vulnerability detection, patterns of unit test, blockchain

I. INTRODUCTION

The blockchain is a public ledger that holds multiple transactions from anonymous parties in blocks [1]. Nowadays, smart contracts are a place for investors and anyone who holds cryptocurrencies. In such a transparent ecosystem, malicious parties seek the vulnerabilities of smart contracts [1]. The developers have used the Solidity programming languages within the development phase. The Solidity languages have significant opcodes for transferring funds or checking authorization [2-3]. If the developers are unaware of the security considerations of such opcodes, the smart contract might be vulnerable, and malicious parties can prepare an attack to get smart contract funds [1]. The DAO hack is a famous example of that. A malicious attacker stole all the funds of the DAO contract by the *call* opcode [1].

There are lots of works in the scope of static and dynamic vulnerabilities detection [4-7]. To the best of our knowledge, all the proposed research is related to automatic detection, and none of them considers modular testing for detecting smart contract vulnerabilities. Most of the previous works are defined in the automatic detection scope, and there is less research on modular testing. Also, developers of smart contracts need modular testing to detect potential vulnerabilities within the development phase. The developers need to apply the detection core to their defined smart contracts business logic. Previous works such as Oyente [4], Mythril [5], and both versions of Securify [6] did not provide modular testing, and they focus on automatic detection. Also, they do not care about the contract business logic within the detection phase. Also, they do not consider smart contract interactions.

We found that only Openzepplin implements a library for assertion testing for checking smart contract balances [8]. This library has based on the unit test. As the global term, the unit test brings modular testing to the developers, and the developer can reuse it for several modules [9]. Also, we found that the software patterns are useful solutions for problems within a context [10-11]. For the first time, the pattern as a global solution has proposed by Christopher Alexander [11]. To the best of our knowledge, the unit test patterns can be a solution that brings modular testing to solve the vulnerability problems in the smart contract development phase.

In this paper, we propose a modular approach for detecting smart contract vulnerabilities. The heart and the soul of our approach are the unit test patterns. This paper proposes three unit test patterns that smart contract developers can use to detect the reentrancy [12], phishable tx.origin [13], and missing protection against the signature replay [14]. The unit test patterns consist of two things. The first is a real-world attacker payload that has occurred. The second is a related unit test that launches such a payload to the developer's smart contract and does the interaction. Each unit test pattern consists of Setup, Action, and Check phases in order. During the Setup phase, the developers deploy the target smart contract and all the related smart contracts and do the initialization. In the Action phase, a malicious attacker runs the core of the test. In the Check phase, if the core of the test runs successfully without any rejection, then the target smart contract is vulnerable to such vulnerability. Also, we propose the test-assistant module that contains a local repository of the unit test and the related payloads. The test-assistant module has a drag and drops script that developers, via the script, import the desired unit test to the Hardhat environment [15]. Next, developers construct their test with the critical aspect. Finally, the developers can execute the test via the Hardhat environment.

To evaluate our proposed approach, we collect a set of vulnerable and non-vulnerable smart contracts as a new benchmark from the Ethereum blockchain, apply the unit test patterns and catch the result. As a comparison, we choose two updated symbolic execution tools and analyze the correctness of the result proposed by symbolic execution tools. The comparison result shows that the modular approach is more effective than symbolic execution tools. Also, it detects vulnerability within the complex implementation.

We believe the proposed approach can be helpful in the smart contract development phase. It also helps developers to create complex implementations.

The rest of the paper is organized as follows: Section II reviews the related work. In Section III, we introduce some preliminaries. Section IV introduces our proposed modular approach and the unit test patterns. Finally, we evaluate the proposed approach in Section V.

II. RELATED WORK

Many works exist that propose static and dynamic smart contract vulnerabilities detection [4-7]. The static analysis refers to the structural analysis. In contrast, dynamic analysis refers to running a solidity program within the blockchain environment [16]. The primary difference between static and dynamic as a type of detection refers to execution. Major research in static and dynamic detection has related to automatic detection [16]. Each of the detection tools in such scopes has an analysis method. The analysis method refers to a major category such as symbolic execution, model checking, constraint solving, etc. Also, automatic detection uses code transformations. A code transformation mechanism could be an abstract syntax tree (AST) transformation, control flow graph (CFG) transformation, and other related mechanisms [16].

Lots of research on static analysis has been conducted. In 2016, the Oyente was the starting point of the vulnerabilities research [4]. The analysis method of the Oyente is related to symbolic execution. It uses the solidity compiler to obtain the Ethereum virtual machine (EVM) bytecode and disassembler from the Go-Ethereum to display the smart contract opcode in the symbolic form. Oyente only captures the reentrancy vulnerability. Several projects, such as Gasper [17] and Ether* (s-gram) [18], use the Oyente as a component. Other projects such as Osiris [19] and MAIAN [20] extend the Oyente approach as further research. Also, the EthIR proposes a modified version of the Oyente [21].

The main idea of all the mentioned tools is to use symbolic execution as a detection core. They lead the research to automatic detection. In 2018, the Mythril was proposed by ConsenSys [5]. Mythril is a symbolic execution engine that translates bytecode into SMT queries of solver Z3 to check for security vulnerabilities. The SMT queries are usually complex to solve, hence the time or other dependencies. The Mythril does not bring modular testing to the developers via the proposed tool and leads the research to automatic detection. In 2018, the SmartCheck was proposed in academic research [22]. It flags the smart contract vulnerabilities via the repositories of the patterns. It converts the contract code into an XML syntax tree. Via the XQuery, the path expression finds the potential vulnerabilities. The SmartCheck does not bring modular testing to the developers via the proposed tool. In 2018, Securify 1.0 was proposed by the ETH Zurich [6]. Securify takes the contract bytecode and security properties as inputs. It decompiles the bytecode and represents the contract code as Datalog facts. The security properties are represented as DataLog rules. Securify uses *Soufflé* as a static analysis synthesis tool to check the facts and guarantee that the patterns are detected.

Also, Securify 2.0 was released in 2020 and supported 37 vulnerabilities [23]. The securify 1.0 and securify 2.0 do not bring modular testing via the proposed tool to developers and lead the research to automatic detection. In 2019, The Trail of Bits proposed the Slither as a static analysis framework [24]. Slither via the solidity compiler, convert smart contract code to AST, and via the Slither core, detects potential vulnerabilities. The Slither core contains information recovery, SlithIR conversion, and code analysis components. Slither does not bring modular testing to developers via the proposed tool and leads the research to automatic detection.

Furthermore, major research on dynamic analysis was proposed. In 2018, ReGuard as dynamic fuzz testing was proposed [25]. It only detects reentrancy vulnerability. The ReGuard has a web interface that takes a solidity code or the bytecode from the user. It translates it to C++ program via the intermediate representation (code to AST, bytecode to CFG). Until now, this tool does not disclose. In 2018, teEther as a new dynamic analysis tool was proposed [7]. The teEther gets the contract bytecode, generates exploits, and verifies such vulnerabilities that cause a payout to arbitrary addresses through SMT solver Z3. The teEther does not bring modular testing to the developers via the proposed tool. In 2018, the MAIAN extended the approach of Oyente and considered that each intended attack needs multiple transactions [20]. Also, the MAIAN is related to symbolic execution, and the detection core is similar to Oyente. The MAIAN also detects reentrancy and self-destruct vulnerabilities. Oyente and the MAIAN use the SMT solver Z3 as a constraint solving engine.

As the main problem, major research in the scope of static and dynamic smart contract vulnerabilities detection leads research to automatic detection. Developers of smart contracts need modular testing to detect potential vulnerabilities within the development phase. The developers need to apply the detection core to their defined smart contracts business logic. Moreover, none of the research considers modular testing and the needs of developers. Furthermore, business logic is not considered in all mentioned research.

As a part of the view, modular testing is related to the unit test that brings tests on the core of the smart contracts business logic [9]. To the best of our knowledge, we found that the software pattern can bring a solution to the problem within a context. The smart contract vulnerabilities are the largest domain that can separate in the classified context. The SWC documents show smart contract weakness classification [26]. We proposed unit test patterns as a new solution that can be helpful for modular testing to solve the developer's needs.

We found that only the Openzepplin organization has implemented the npm package library for assertion testing for checking smart contract balances of smart contracts uint tests [8]. In 2021, Wöhrer and Zdun proposed research around the DevOps for Ethereum blockchain smart contracts [9]. The proposed paper brings the typical stages and activities in DevOps that blockchain developers use. The research shows that the unit test is a white box testing mechanism related to the dynamic testing scope that can be one of the DevOps activities.

In this paper, we observe that much of the research leads developers to use automatic testing. In contrast, the developers need the modular approach for checking the implementation and detecting potential vulnerabilities on their own. Also, the major smart contracts interact together and serve the service to parties. Automatic detection does not consider the interaction.

The modular approach that supports the unit test patterns as a key element of smart contract vulnerability detection can resolve the mentioned problem. The developers can check the smart contract vulnerabilities within the development phase by using such a mechanism.

III. PRELIMINARIES

A. *Unit test as white box testing mechanism*

This section briefly introduces the unit test as a white box testing mechanism. The testing scope is divided into static and dynamic testing [9]. One of the branches of static testing is static analysis. Static analysis is a mechanism related to formal verification [9]. Static analysis tools get smart contract source code or related bytecode and convert it to any desired

intermediate representation. The intermediate representation is the model that arrives at formal verification tools [9, 16].

In contrast, dynamic testing is divided into functional and non-functional testing [9]. On functional testing scope, two mechanisms are defined, the first one is white box testing, and the second one is black box testing [9]. Unit, integration, and mutation tests are related to white box testing. In contrast, the system, user acceptant, and fuzz tests are related to black box testing [9]. At the white box testing, the developers fully know the internal steps of codes and do the test. In contrast, a user does not know the functionality in black box testing and does a test. Unit test is a full-step mechanism that smart contract developers can use within the development phase [9].

B. The vulnerability root cause as a pattern

For the first time, the pattern in the architectural domain was proposed by Christopher Alexander [11]. Nowadays, patterns are standard solutions in software engineering. Patterns are solutions to repetitive problems within a context. In software engineering, the expert proposes the patterns. Others clones and uses them to solve a problem within a domain [10-11]. In the abstract view, a smart contract vulnerability has major attack vectors related to a root cause [5, 26]. For example, reentrancy vulnerability has major attack vectors caused by an attack to re-enter a payout function due to the transferring Ether opcode. In a payout function, a vulnerable *call* opcode can lead a smart contract to reentrancy vulnerability [1, 12]. Such root cause can be summerize to a pattern. The mentioned example is related to a real-world case DAO [1]. This article proposes unit test patterns as called reveal reentrancy for such examples. The vulnerability root cause can be digested as a simple pattern for detection. Also, each pattern has a template called pattern form. Pattern form introduces the face of a pattern [10]. Developers can apply and construct the pattern as their implementation.

C. Gas Station Network (GSN)

In the Ethereum blockchain, anyone who sends Ethereum transactions needs a few Ethers to pay the transaction gas fees. This forces new users to purchase a few Ethers for broadcasting their transactions to the Ethereum network. This problem was solved by defining Gas Station Network (GSN) [27]. Each GSN as a hole business domain contains a few relayers (third-party nodes). The user sends the gasless transaction to a relayer. The relayer broadcasts the transaction to the Ethereum network and pays its gas fee [27].

D. Meta-Transaction

Gas of the Ethereum is a reward of execution that the miners take. Nowadays, the Ethereum gas fee increases, and participants must pay the higher execution fees. This problem was solved by proposing Meta-Transaction [27]. Each Ethereum transaction consists of a smart contract's input parameters. However, the meta-transaction includes further information like the signature of the input parameters. Input parameters are input arguments of a function from the smart contract. The GSN is a solution that helps broadcast the meta-transaction to the Ethereum network. The authorized party of a smart contract sends gasless meta-transaction to the relayer. A relayer pays the transaction gas fee and broadcasts such transactions to the Ethereum network [27].

IV. The Proposed Modular Testing Approach

In this section, we propose our modular testing approach and show the usage process. We introduce our three unit test patterns, and in the next section, we propose our evaluation and comparison results.

A. The usage process

The Hardhat is an Ethereum local blockchain environment. The developers can build smart contracts through such an environment and execute unit tests or integrate smart contracts into Ethereum networks. By default, Hardhat runs the smart contracts on the local blockchain.

The smart contract contains critical aspects such as payout and payable. The contract with the payable aspect captures the Ether from the outside world. In contrast, the payout aspect transfers Ether to the outside. The usage process of our approach is presented in Fig.1.

Fig.1 shows the following steps:

1. The developer extracts such mentioned aspects from the smart contracts within the development phase.
2. The developer picks a unit test pattern by executing the test-assistant script.
3. The test-assistant module inserts the unit test and the related contract payload into the Hardhat blockchain environment.
4. The developer adjusts the business logic and pushes the critical aspect that the unit test patterns need.
5. The developer executes the modular test and views the result in the console. In the final step, the unit test patterns observe the vulnerability.

Also, the Hardhat environment executes the interaction between the payload contract and the developer's smart contracts. Our modular approach is related to dynamic testing that uses the critical aspect for observing the vulnerabilities. The critical aspect is the essential aspect that unit test patterns need for interaction.

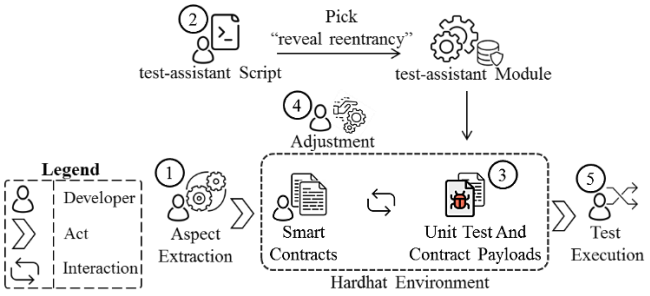

Fig. 1. The process of using the modular approach

Our test-assistant module contains a local repository with our proposed three unit test patterns. The test-assistant module looks like a test grabber that contains the assistant library for further tests. Our approach and research artifacts are available on the Google drive repository [28].

B. Unit test patterns

In this part, we propose three unit test patterns that developers can use for observing vulnerability.

1) Reveal reentrancy

The reveal reentrancy detects reentrancy vulnerability within smart contracts. It requires a payable and payout aspect that contains transferring opcode. The pattern form is as follows:

- *SWC-ID:* SWC-107
- *Pattern Name:* Reveal reentrancy
- *Intent:* provides a prototype for detecting reentrancy vulnerability.

- *Background:* Gas is a consuming unit of Solidity opcode that EVM use. Solidity has three different transferring Ether opcodes. Each transferring opcode has a forwarding gas execution amount. *Transfer* or *send* opcodes forward 2300 gas while the *call* opcode forwards all the gas. The smart contract might be vulnerable to reentrancy if it forwards more than 2300 gas to a malicious callee [12].
- *Historical Case:* The DAO hack occurred in June 2016 [1]. An anonymous attacker found a reentrancy bug within the DAO contract code. In the first few hours of the attack, about 3.6 million Ethers were stolen. The attacker re-enters a payout function and withdraws funds multiple times before the smart contract can update its balance.
- *Root Cause:* Forwarding extra gas to a callee. The callee could be a malicious smart contract [12].
- *Problem:* A malicious contract can withdraw all the contract balances if the contract implements a payout function with a *call* opcode and does not use the reentrancy guard (Semaphor).
- *Applicability:* Use the reveal reentrancy on a payout aspect that contains a transferring Ether opcode. The payout aspect is the withdrawal function of a smart contract.
- *Interaction Protocol:* The protocol in order contains three phases that the developer can construct. In this protocol, the environment externally owned accounts (EOAs) are defined as the Owner, Attacker, Alice, and Bob.

  Setup phase:
  1. The Owner deploys the target contract and all related contracts.
  2. The Attacker deploys the Reveal_reentrancy contract.

  Action phase:
  3. The non-malicious parties, such as Alice and Bob, pay the target contract.
  4. The Attacker, via the Reveal_reentrancy contract, tries the withdrawal reentrancy on the target contract.

  Check phase:
  5. If the Reveal_reentrancy contract does the reentrancy successful, then the target contract is vulnerable.
- *Relation With Other Vulnerabilities:*

  Message call with the hardcoded gas amount (SWC-134) [26]: Due to the Ethereum hard fork issues, the SWC documents advise developers to use the *call* opcode instead of the *send* or *transfer*. The smart contract might be vulnerable to reentrancy by using *call* opcode in a different code segment.
- *Proof Of Concept:* Reveal reentrancy detects major vulnerable smart contracts from our benchmark (see Section V).
- *Motivation Example And The Related Unit Test:* Refer to our catalog [28].
- *The Core Of Detection:* The mechanism of the reveal reentrancy checks the rejection of execution flow. Suppose a smart contract breaks the execution flow of re-entering a callee with any proper mechanism. In that case, the attacker contract cannot re-enter the smart contract for a while.

2) Reveal origin

The reveal origin detects tx.origin vulnerability within a smart contract. It requires payable and other aspects that contain *tx.origin* opcode. The pattern form is as follows:

- *SWC-ID:* SWC-115
- *Pattern Name:* Reveal origin
- *Intent:* Reveal origin provides a prototype for detecting tx.origin vulnerability.
- *Background:* The tx.origin is a global variable in solidity, which refers to an account's address that initiates the transaction. In contrast, the msg.sender is a global variable that gives the direct sender of the message [13]. Fig. 2 shows the difference while party and smart contracts call each other.

Fig. 2. The difference between msg.sender and tx.origin

- *Historical Case:* THORChain was a smart contract that implemented the *RUNE* token [29]. CoinEx team discovers tx.origin vulnerabilities in the THORChain smart contract and releases a report [30]. The report shows that the developer of the THORChain makes a mistake on the *transferTo* function. This function implements the transferring token functionality. In the correct implementation, the *transferTo* must send the amount of token from the current sender to a recipient. In contrast, the code shows that the sender has declared as tx.origin instead of the msg.sender.

```
function transferTo(address recipient, uint256 amount) public returns (bool){
    _transfer(msg.sender, recipient, amount);
    return true;
}
```

Fig. 3. The correct implementation of THORChain

```
function transferTo(address recipient, uint256 amount) public returns (bool){
    _transfer(tx.origin, recipient, amount);
    return true;
}
```

Fig. 4. The wrong implementation of THORChain

Fig. 3 shows the correct implementation of the *transferTo* function from the THORChain smart contract. In contrast, Fig. 4 shows the wrong implementation. Also, the report shows that a malicious attacker can hijack the call from the origin (*RUNE* token holder), run the *transferTo* with the desired amount, and steal the tokens.

- *Root Cause:* Improper access control [13].
- *Problem:* If a smart contract implements the wrong authorization by using the tx.origin, the malicious attacker can pass the authorization by hijacking and leading the call of the origin (contract owners) to the smart contract. A malicious attacker can trick the origin into a trustworthy action, hijack the origin's call, and leads the call into the smart contract. In a further step, the smart contract authorizes the origin, and the malicious attacker can run a functionality. The tx.origin vulnerability can cause a phishing attack. So the name is called phishable tx.origin.
- *Applicability:* Use the reveal origin on a payout or other aspect that contains a *tx.origin* opcode.

- *Interaction Protocol:* The protocol in order contains three phases that the developer can construct. In this protocol, the environment EOAs are defined as the Owner, Attacker, Alice, and Bob.
  Setup phase:
  1. The Owner deploys the target contract and all related contracts.
  2. The Attacker deployed the Reveal_origin contract and the mask contracts. Wallet and Log are the mask contracts. Instead of the Log contract address within the Wallet contract, the Attacker sets the Reveal_origin's address.

  Action phase:
  3. The non-malicious parties, such as Alice and Bob, pay the target contract.
  4. The Attacker tricks the Owner into performing trustworthy actions like transferring a few Ethers to the Wallet contract. Reveal_origin contract leads the Owner's call to the target contract and tries to bypass the authorization.

  Check phase:
  5. If the authorization does pass successfully, the contract is vulnerable.
- *Relation With Other Vulnerabilities:* -
- *Proof Of Concept:* Reveal origin detects major vulnerable smart contracts from our benchmark (see Section V).
- *Motivation Example And The Related Unit Test:* Refer to our catalog [28].
- *The Core Of Detection:* The reveal origin mechanism checks the authorization. If the authorization does pass successfully, the contract is vulnerable to tx.origin vulnerability. As the key detection concept, the few implementations of the tx.origin is non-vulnerable. In Solidity programming, there is no mechanism to check that the sender of a message is a contract or EOA. Developer checks address of non-contract account via the using of tx.origin. Fig. 5. shows the mentioned situation. In the EVM stack, the origin variable cannot be a contract address. The developer also checks that the origin's address must be the sender's address.

```
// origin of the transaction must be an owner, and the origin of the
// transaction must be a sender (so the owner is a msg.sender)
require(tx.origin = = owner && tx.origin = = msg.sender);
```

Fig. 5. Check the non-contract account address via the tx.origin

3) *Reveal missing protection against the signature replay (short name: reveal signature replay)*

The reveal signature replay detects missing protection against the signature replay vulnerability within smart contracts. It requires the aspect that contains the *signature* opcode. The pattern form is as follows:

- *SWC-ID:* SWC-121
- *Pattern Name:* Reveal missing protection against the signature replay (short name: reveal signature replay)
- *Intent:* Reveal signature replay provides a prototype for detecting missing protection against the signature replay.
- *Background:* The previous section introduces the meta-transaction that brings the gasless transaction to the parties. Before sending a meta transaction, the authorized party of a smart contract signs the hash of the input parameters with its private key. Other parties do not have the private key of such an authorized party. After sending meta-transaction, the smart contract catches the request, calculates the input parameters hash, and extracts the signer public key from the receiving sign via the EIP-712 standard [31]. The contract executes the requesting functionality if the extracted signer public key is the same as the authorized party.
- *Historical Case:* The Civil is a decentralized and censorship-resistant ecosystem for online journalism [32]. The Civil ecosystem contains a *Newsroom* smart contract. The contract includes *pushRevision* and *verifiyRevisionSignature* functions as the main functionality. The *pushRevision* function handles updates or revisions to existing content and *verifiyRevisionSignature* with the proposed revision and the content author who initially created the first signed revision. A malicious author could take valid signatures and content hashes from another author and create a new valid revision without their knowledge. Civil has fixed the issue by rejecting hashes already part of the previous revision.
- *Root cause:* Improper protection against signature replay [14].
- *Problem:* If a smart contract does not implement a one-time signature, a malicious party can take and replay the input parameters of a function with the related sign to the contract. Finally, the contract verifies the signer's signature and does a task for a malicious party.
- *Applicability:* Use the reveal signature replay on any functionality aspect that contains a *signature* opcode and the hashes of the input parameters.
- *Interaction Protocol:* The protocol in order contains three phases that the developers can construct. In this protocol, the environment EOAs are defined as the Owner, Attacker, Alice, and Bob.
  Setup phase:
  1. The Owner deploys the target contract and all related contracts.

  Action phase:
  2. The non-malicious parties, such as Alice and Bob, pay the target contract.
  3. The Owner does the following:
     a) Get the hash of the input parameters.
     b) Make a transaction by signing the previous hash with its private key.
     c) Send the transaction to the target contract via the relayer.
  4. The Attacker clones the transaction.
  5. The Attacker replays the cloned transaction to the target contract via the relayer.

  Check phase:
  6. If the contract does a task without rejecting authorization, then the contract is vulnerable.

- *Relation With Other Vulnerabilities:* -

- *Proof Of Concept:* Reveal signature replay detects major vulnerable smart contracts from our benchmark (see Section V).
- *Motivation Example And The Related Unit Test:* Refer to our catalog [28].
- *The Core Of Detection:* The mechanism of the reveal signature replay checks the contract protection against the signature replay. If the contract does a task without rejecting authorization, then the contract is vulnerable to the missing protection against the signature replay. The core of detection does base on an act and reaction. The attacker only mimics the action of an owner or an authorized party.

V. EVALUATION

In this research, the proposed approach is evaluated in terms of effectiveness and performance. We propose two research questions as follows:

RQ1: Is the modular approach effective than automatic detection tools?

RQ2: How efficient is the modular approach in this research compared to automatic detection tools?

To answer the first research question, we collect major smart contracts from the Ethereum blockchain classified as vulnerable and non-vulnerable. Table I. shows the smart contracts benchmark.

TABLE I. THE SMART CONTRACTS BENCHMARK

Num.	Detection Goal	Contract Name	Results		
1	Reentrancy	coinwallet	FP	FP	TD
2		Ethtragon	FP	FP	TD
3		Hostpay	FP	FP	TD
4		userwallet, titan	NA	FP	TD
5		walleti	FP	FP	TD
6		CrossReentrancy	TD	TD	TD
7		EtherBank	TD	TD	TD
8		EtherCoin, EtherPool	FN	FN	TD
9		EtherStore	TD	TD	TD
10		Graffiti	TD	TD	TD
11		ModifierEntrancy	TD	TD	TD
12		Origin, Agent	TD	TD	TD
13		PrivateBank	TD	TD	TD
14		Reentrance	TD	TD	TD
15		Reentrancy_bonus	TD	TD	TD
16		ReentrancyDAO	TD	TD	TD
17		SimpleDAO	TD	TD	TD
18	Missing protection against signature replay	GrantFlow	TD	TD	TD
19		ZEDwallet	NA	TD	TD
20		CryptoMif	FN	FN	TD
21		Thrifty	FN	FN	TD
22	Phishable tx.origin	Cryptoary	FP	TD	TD
23		EasterEgg	NA	TD	TD
24		Ethemerce	FN	FN	TD
25		Primary, Crawler	FN	FN	TD
26		CryptoDaemon	TD	TD	TD
27		Coinic	FN	FN	TD
28		Bridge, AirDrop	FN	FN	TD

Legend
Result in order: Mythril, Slither, Our approach
Gray row: Vulnerable case White row: Non-vulnerable case
FP: False positive FN: False negative
TD: True detection NA: Not detect within 15 min.

These smart contracts bring a new benchmark. We collect a few vulnerable smart contracts via Karl. The Karl is a real-time tool that can monitor the Ethereum blockchain for newly deployed vulnerable smart contracts. Other smart contracts via the inspection are gathered, or a research source was proposed.

Also, The proposed benchmark contains the twice smart contracts that interact and work together. Other researches benchmark was not provided in such cases [33-34]. As a key detection concept, smart contracts in blockchain environments can interact to provide a service to EOAs [9].

The first seventeen smart contracts contain transferring Ether opcodes or, in a specific case, does an assignment of tokens (modifierEntrancy). We declare the detection goal for such cases and obtain a unit test on such functionality. The interaction protocol is the same within the test and leads the developers to declare further interactions.

On benchmark, the test result from our approaches generates true detection. Also, we choose Mythril and Slither as updated tools that can detect major vulnerabilities such as reentrancy, tx.origin, etc.

As a comparison, we import smart contracts from our benchmark to the Mythril and Slither. Finally, we analyzed the correctness of the tools reports that consider the goal of detecting such functionality related to our test. We reflect all results in Table I.

Also, we write the correctness of the tools report as further analysis. The correctness was written at the end of the tools report. Furthermore, other research mentioned that the Mythril and Slither as updated tools that might lead a test to a false positive or negative [33].

To answer the second research question, we found that each detection tool needs compiler support to get the intermediate representation model. For example, securify 2.0 only detects smart contracts that the pragma version ranges from 0.5.8 to 0.7.0 [23]. To the best of our knowledge, due to the security consideration, the solidity document leads the developers to use the latest compiler pragma. While securify 2.0 cannot detect such cases. Also, the SmartCheck cannot detect contracts pragma upper than 0.6.0 [35]. On the other hand, the Hardhat environment supports all compiler pragma, and our approach is based on the Hardhat environment.

Furthermore, it is the reusability of our modular approach. The developer can reuse unit tests as further needs for different components used within a project. Our test-assistant module has a service for getting a new uint test pattern and the expert's related payload. Such unit test patterns insert in a local repository of the test-assistant module. These are the two metrics that show the performance in action.

VI. CONCLUSION

In this paper, we propose a modular testing approach for smart contract vulnerability detections. In contrast to automatic detection, the developers need a modular testing mechanism to detect smart contract vulnerabilities within the development phase. One solution is to provide the developer with modular testing based on unit test patterns. To achieve such a solution, this research proposed a dynamic approach in which developers use unit test patterns as a white box mechanism to observe the vulnerabilities. Also, this research proposed three unit test patterns that detect reentrancy, phishable tx.origin, and missing protection against signature replay vulnerabilities. For evaluation, a new benchmark that contains single and twice Ethereum smart contracts that are vulnerable and non-vulnerable is proposed. In this way, two of the updated automatic detection tools are chosen for comparison. After all, the correctness of the tools report is analyzed. Also, the modular approach can help developers to be used the latest smart contracts. The proposed approach can

be applied in the development phase and observe the smart contract vulnerabilities. As future work in the research direction, we intend to extend the modular approach into a modular testing framework that considers unit test patterns as root vulnerability observers.

REFERENCES

[1] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A Survey on Ethereum Systems Security," ACM Computing Surveys, vol. 53, no. 3. Association for Computing Machinery (ACM), pp. 1–43, May 31, 2021. doi: 10.1145/3391195.

[2] G. Zheng, L. Gao, L. Huang, and J. Guan, "Solidity Basics," Ethereum Smart Contract Development in Solidity. Springer Singapore, pp. 49–83, Sep. 01, 2020. doi: 10.1007/978-981-15-6218-1_3.

[3] G. Wood, "ETHEREUM: A secure decentralised generalised transaction ledger," Tech. Rep., 2014, Accessed: Jun. 05, 2022. [Online]. Available: http://gavwood.com/Paper.pdf

[4] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, Oct. 24, 2016. doi: 10.1145/2976749.2978309.

[5] B. Mueller, "Smashing Ethereum Smart Contracts for Fun and Real Profit," in 9th HITB Security Conference, 2018.

[6] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM, Jan. 15, 2018. doi: 10.1145/3243734.3243780.

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

[8] E. Lau, F. Giordano, and H. Croubois, "Test helpers," Openzeppelin.com. [Online]. Available: https://docs.openzeppelin.com/test-helpers/0.5/. [Accessed: 05-Jun-2022].

[9] M. Wohrer and U. Zdun, "DevOps for Ethereum Blockchain Smart Contracts," 2021 IEEE International Conference on Blockchain (Blockchain). IEEE, Dec. 2021. doi: 10.1109/blockchain53845.2021.00040.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design patterns: Elements of reusable object-oriented software. Boston, MA: Addison Wesley, 1994.

[11] M. Fowler, Analysis patterns: Reusable object models. Boston, MA: Addison Wesley, 1996.

[12] P. Collins, "Reentrancy (SWC-107)," swcregistry.io. [Online]. Available: https://swcregistry.io/docs/SWC-107. [Accessed: 05-Jun-2022].

[13] P. Collins, "Authorization through tx.origin (SWC-115)," swcregistry.io. [Online]. Available: https://swcregistry.io/docs/SWC-115. [Accessed: 05-Jun-2022].

[14] P. Collins, "Missing Protection against Signature Replay Attacks (SWC-121)," swcregistry.io. [Online]. Available: https://swcregistry.io/docs/SWC-121. [Accessed: 05-Jun-2022].

[15] F. Victorio, F. Zeoli, P. Palladino, and J. Kane, "Hardhat," Ethereum development environment for professionals. [Online]. Available: https://hardhat.org/. [Accessed: 05-Jun-2022].

[16] M. di Angelo and G. Salzer, "A Survey of Tools for Analyzing Ethereum Smart Contracts," 2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON). IEEE, Apr. 2019. doi: 10.1109/dappcon.2019.00018.

[17] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, Feb. 2017. doi: 10.1109/saner.2017.7884650.

[18] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: towards semantic-aware security auditing for Ethereum smart contracts," Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ACM, Sep. 03, 2018. doi: 10.1145/3238147.3240728.

[19] C. F. Torres, J. Schütte, and R. State, "Osiris," Proceedings of the 34th Annual Computer Security Applications Conference. ACM, Dec. 03, 2018. doi: 10.1145/3274694.3274737.

[20] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale," Proceedings of the 34th Annual Computer Security Applications Conference. ACM, Dec. 03, 2018. doi: 10.1145/3274694.3274743.

[21] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "EthIR: A Framework for High-Level Analysis of Ethereum Bytecode," Automated Technology for Verification and Analysis. Springer International Publishing, pp. 513–520, 2018. doi: 10.1007/978-3-030-01090-4_30.

[22] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko and Y. Alexandrov, "SmartCheck: Static Analysis of Ethereum Smart Contracts," 2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB), 2018, pp. 9-16.

[23] P. Tsankov, Y. Sachinoglou, and C. Nuckols, "Securify v2.0." [Online]. Available: https://github.com/eth-sri/securify2. [Accessed: 05-Jun-2022].

[24] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," 2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB). IEEE, May 2019. doi: 10.1109/wetseb.2019.00008.

[25] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen and B. Roscoe, "ReGuard: Finding Reentrancy Bugs in Smart Contracts," 2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion), 2018, pp. 65-68.

[26] P. Collins, "Smart Contract Weakness Classification and Test Cases," SWC Registry. [Online]. Available: https://swcregistry.io. [Accessed: 01-Jun-2022].

[27] I. A. Seres, "On Blockchain Metatransactions," 2020 IEEE International Conference on Blockchain (Blockchain). IEEE, Nov. 2020. doi: 10.1109/blockchain50366.2020.00029.

[28] M. R. Talebi, "research artifacts" [Online]. Available: https://drive.google.com/file/d/1rF9Nw5XeCVZ33AzJANUuRumzpKJLtZpT/view?usp=sharing. [Accessed: 05-Jun-2022].

[29] THORChain, "The smart contract of THORChain," Etherscan.io. [Online]. Available: https://etherscan.io/address/0x3155ba85d5f96b2d030a4966af206230e46849cb#code#L164. [Accessed: 01-Jun-2022].

[30] CoinEx, "The Security Risks of THORChain," Coinex.com. [Online]. Available: https://announcement.coinex.com/hc/en-us/articles/6175458479252-CoinEx-Monthly-Report-April-2022-. [Accessed: 01-Jun-2022].

[31] R. Bloemen, L. Logvinov, and J. Evans, "EIP-712: Ethereum typed structured data hashing and signing," Ethereum Improvement Proposals, 12-Sep-2017. [Online]. Available: https://eips.ethereum.org/EIPS/eip-712. [Accessed: 05-Jun-2022].

[32] G. Wagner, "Discovering signature verification bugs in ethereum smart contracts," ConsenSys Media, 11-Oct-2018. [Online]. Available: https://media.consensys.net/discovering-signature-verification-bugs-in-ethereum-smart-contracts-424a494c6585. [Accessed: 01-Jun-2022].

[33] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, Jul. 13, 2020. doi: 10.1145/3395363.3397385.

[34] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "SmartBugs: a framework to analyze solidity smart contracts," Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. ACM, Dec. 21, 2020. doi: 10.1145/3324884.3415298.

[35] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck – a static analysis tool that detects vulnerabilities and bugs in Solidity programs (Ethereum-based smart contracts)." [Online]. Available: https://github.com/smartdec/smartcheck. [Accessed: 05-Jun-2022].