Reentrancy Vulnerability Identification in Ethereum Smart Contracts

Noama Fatima Samreen, Manar H. Alalfi
Department of Computer Science
Ryerson University, Toronto, ON, Canada
{noama.samreen,manar.alalfi}@ryerson.ca

***Abstract*—Ethereum Smart contracts use blockchain to transfer values among peers on networks without central agency. These programs are deployed on decentralized applications running on top of the blockchain consensus protocol to enable people make agreements in a transparent and conflict free environment. The security vulnerabilities within those smart contracts are a potential threat to the applications and have caused huge financial losses to their users. In this paper, we present a framework that combines static and dynamic analysis to detect *Reentrancy* vulnerabilities in Ethereum smart contracts. This framework generates an attacker contract based on the ABI specifications of smart contracts under test and analyzes the contract interaction to precisely report *Reentrancy* vulnerability. We conducted a preliminary evaluation of our proposed framework on 5 modified smart contracts from Etherscan and our framework was able to detect the *Reentrancy* vulnerability in all our modified contracts. Our framework analyzes smart contracts statically to identify potentially vulnerable functions and then uses dynamic analysis to precisely confirm *Reentrancy* vulnerability, thus achieving increased performance and reduced false positives.**

I. Introduction

Blockchain technology(BT) has been gaining popularity because of its wide range of potential applications. It was first applied as a cryptocurrency, called Bitcoin [1], but has since been used in many other applications such as e-commerce, trade and commerce, production and manufacturing, banking, and gaming. BT uses a peer-to-peer (peers are known as miners in BT) framework which is a more decentralized approach to storing transaction and data registers. As there is no single point of failure or a third-party centralized control of transactions, BT has been standing out from other cryptocurrency technologies. It uses a chain of blocks in which each block is locked cryptographically using the hash of the previous block it is linked to, which creates an immutable database of all transactions stored as a digital ledger, and it cannot be changed without affecting all the blocks linked together in the chain [2].

Manipulations to the Blockchain is done using a proof of work (POW) system in which computers must solve a complex computational math problem to become eligible to add a block to the Blockchain. The theoretical aspect of using a cryptographically locked chain of blocks may seem to ensure data security and integrity from unauthorized access. However, on-going research on the security, integrity and authenticity of BT have shown that many applications have been exposed to an intrusion attack [3].

One of the most destructive attacks in Solidity smart contract is *Reentrancy* attacks. A *Reentrancy* attack occurs when the attacker drains funds from the target by recursively calling the target's withdraw function. When the contract fails to update its state, a victim's balance, prior to sending funds, the attacker can continuously call the withdraw function to drain the contract's funds. A famous real-world *Reentrancy* attack is the DAO attack which caused a loss of 60 million US dollars.

According to research by Liu et al. [4] detecting the existence of *Reentrancy* vulnerabilities in Smart-Contracts faces two challenges:

- The implementation of Smart-Contracts varies widely and analysis to account for all possible scenarios may become infeasible.
- The *Reentrancy* vulnerability cannot be detected accurately because it lacks a definite pattern in the context of Smart-Contracts. Analyzing with a straightforward and simple pattern may result in false positives, while precise and rigorous patterns may fail to report the existence of a *Reentrancy* vulnerability.

This paper presents a framework to address the *Reentrancy* attack and provide solutions for the above challenges. This paper has the following contributions:

1) A semi-automated framework with a combined static and dynamic analysis to better capture the various patterns of the *Reentrancy* attack efficiently and accurately.
2) A demonstration of the framework on the analysis of 5 smart contracts and comparison with one of the state of the art tools.
3) A solidity smart contract TXL grammar which provides a fixable means to analyze various implementations and versions of smart contracts.

II. Background

A. Ethereum

One of the Blockchain technology platforms is Ethereum [5] which can implement algorithms expressed in a general-purpose programming language allowing developers to build a variety of applications, ranging from simple wallet applications to complex financial systems for the banking industry. These programs are known as Smart-Contracts which are written in a Turing-complete bytecode language, called EVM bytecode [5]. The transactions sent to the Ethereum network by the users can create new contracts, invoke functions of a contract, and/or transfer ether to contracts.

B. Smart Contract

Ethereum uses Smart-Contracts [5], which are computer programs that directly controls the flow or transfer of digital assets. Each function invocation in a Smart-Contract is executed by all miners in the Ethereum network and they receive execution fees paid by the users for it. Execution fees also protect against denial-of-service attacks, where an attacker tries to slow down the network by requesting time-consuming computations. This execution fee is defined as a product of "gas" and "gas-price". Implementing a Smart-Contract use case can pose few security challenges, like public visibility of the complete source code of an application on a network, and validation and verification of the source code. Moreover, the immutability of Blockchain technology makes patching discovered vulnerabilities in already deployed Smart-Contracts impossible.

```
pragma solidity >=0.4.22 <0.6.0;
contract Sender {
    uint public amount;
    address payable public sender;
    address payable public reciever;
    constructor() public payable {
        sender = msg.sender;
        amount = msg.value;
    }
    function send(receiver) payable {
        receiver.call.value(value).gas(20317)();
    }
}
contract Receiver {
  uint public balance = 0;
  function () payable {
    balance += msg.value;
  }
}
```

Listing 1. Solidity Contracts Sender and Reciever

C. Solidity

Smart contracts are typically written in a high-level Turing-complete programming language such as Solidity [6], and then compiled to the Ethereum Virtual Machine (EVM) bytecode [5], a low-level stack-based language. For instance, Listing 1 shows a smart contract written in the Solidity programming language [6].

In Listing 1, the first line of the program declares the Solidity's version. The program will be compatible with the corresponding EVM or any other higher version less than 0.6.0. It contains a constructor to create an instance of the contract, and functions.

The *msg.sender* is a built-in global variable representative of the address that is calling the function. The *msg.value* is another built-in variable that tells us how much ether has been sent. The *payable* keyword is what makes solidity truly unique. It allows a function to send and receive ether.

A function with no name followed by *payable* keyword, *function () payable*, is a fallback function in solidity that is triggered when a function call's identifier does not match any of the existing functions in a smart contract or if there was no data supplied at all.

To transfer ether between the contracts, Solidity uses *send()*, *transfer()* and *call()*. *send()* transfers the ether and executes the fallback function of the contract that receives ether. The gas limit per execution is 2300 and an unsuccessful execution of *send()* function does not throw an exception but the gas required for the execution is spent. Similarly, *transfer()* is also used to transfer ether between the contract, but the gas limit can be redefined using the *.gas()* modifier, and an unsuccessful *transfer()* throws an exception. The gas limit in both these functions prevents the security risk involved in executing expensive state changing code in the fallback function of the contract receiving the ether. The one pitfall is when a contract sets a custom amount of gas using the *.gas()* modifier. The *call()* function is comparatively more vulnerable as there is no gas limit associated with this function.

III. Reentrancy Vulnerability

According to Atzei et al. [2], a *Reentrancy* attack can drain a Smart-Contract of its ether and can aid an intrusion into the contract code. When an external function call another untrusted contract and an attacker gains control of this untrusted contract, they can make a recursive call back to the original function, unexpectedly repeating transactions that would have otherwise not run, and eventually consume all the gas.

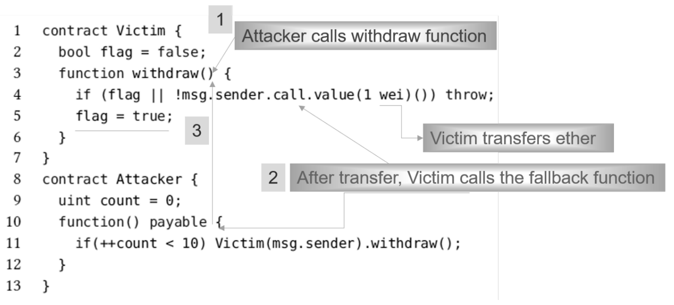
A. On A Single Function

If a contract uses *call*, *send* or *transfer* which may cause control flow to an external contract, with a fallback function, and then updates the state afterward then this causes the state of the contract to be incomplete when flow control is transferred (see Listing 2). Therefore, when this fallback function is triggered, the flow of control may not return as the called contract expects and the caller might do any number of unexpected things such as calling the function again, calling another function or even calling another contract.

```
function transferBalance(address receiver, uint
    amount)
 public {
  require(balances[msg.sender] >= amount);
  receiver.transfer(amount);
  /* flow control transferred before the sender's
      balance is updated and before an event is
      emitted. Potentially the start of trouble. */
  balances[receiver] -= amount;
  LogTransactions(msg.sender,receiver, amount);
  }
```

Listing 2. Reentrancy vulnerability on a single function

In the example given, one of the mitigation methods is to use send() which will limit, to an extent, the execution of expensive external code [7]. However, this can be completely avoided by updating *balances* before transferring the control to another contract. That is, put the whole state in order before invoking another contract. If something goes wrong, revert everything to the original state (see Listing 3).



```
contract Victim {
  bool flag = false;
  function withdraw() {
    if (flag || !msg.sender.call.value(1 wei)()) throw;
    flag = true;
  }
}
contract Attacker {
  uint count = 0;
  function() payable {
    if(++count < 10) Victim(msg.sender).withdraw();
  }
}
```

1
Attacker calls withdraw function
3
Victim transfers ether
2
After transfer, Victim calls the fallback function

Fig. 1. The DAO Attack

```
function transferBalance(address receiver,uint
    amount)public{
    require(balances[msg.sender]>=amount);
    balances[msg.sender]-=amount;
    LogTransactions(msg.sender,receiver, amount);
    receiver.transfer(amount);
    //<==on fail, this will revert all the above.
  }
```

Listing 3. Reentrancy Vulnerability on a single function - Prevention

B. Cross-Function Reentrancy

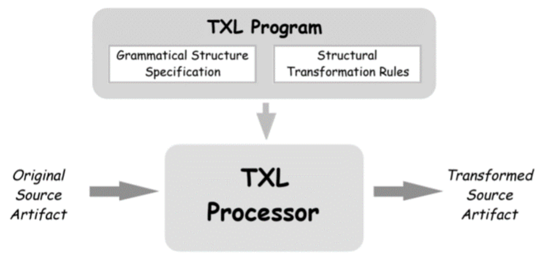
A similar attack can be done when two different functions or even contracts share the same state. In this case (Listing 4), the attacker calls transfer() when their code is executed on the external call in *withdraw*. Since their balance has not yet been set to 0, they are able to transfer the tokens even though they already received the withdrawal. This vulnerability was also used in the DAO attack [3]. The same solution to update all the balances before transferring control to another function or contract will work even in this case.

```
mapping (address => uint) private balance;
function transfer(address to, uint amount) {
    if (balance[msg.sender] >= amount){
      balance[to] += amount;
      balance[msg.sender] -= amount;
    }
  }
function withdraw() public {
   uint amount = balance[msg.sender];
   require(msg.sender.call.value(amount)());
   /* At this point, the caller's code is executed,
       and can call transfer() */
   balance[msg.sender] = 0;
  }
```

Listing 4. Cross-function Reentrancy vulnerability

IV. Motivation

The Decentralized Autonomous Organization (known as the DAO) was initiated in May 2016 as a venture capital fund for the crypto and decentralized space [2], built as a smart contract on Ethereum blockchain concept. The lack of a centralized authority aided in running this organization at a reduced cost and it provided more control and access to the investors that participated. During the creation period of the DAO, anyone could send Ether to a unique wallet address in exchange for DAO tokens. This creation period gathered 12.7M Ether which was worth around USD 150M at the time,



TXL Program
Grammatical Structure Specification
Structural Transformation Rules
Original Source Artifact
TXL Processor
Transformed Source Artifact

Fig. 2. Txl Paradigm [9]

making it a huge success in the Ethereum world. Essentially, this platform was created to fund anyone with a project idea deemed potentially profitable by the community. Anyone with DAO tokens could vote on the pitch and receive rewards in return if the projects turned a profit. However, on June 17, 2016, a hacker was able to attack this Smart contract by exploiting a vulnerability in the code that allowed him to transfer funds from the DAO. As reported by M. Saad et al. [8] approximately, 3.6 million Ether was stolen, the equivalent of USD 70M at the time. The attacker was able to request the smart contract (DAO) to give the Ether back multiple times before the smart contract could update its balance. This was possible because of the fact that when the DAO smart contract was written, the developers did not consider the possibility of a recursive call and the fact that the smart contract first sent the ether and then updated the internal token balance. The reentrancy vulnerability exploitation in the DAO attack(as shown in Figure 1) was accomplished in four steps,

1) The Attacker initiates a transaction by calling withdraw function of Victim;
2) The Victim transfers the money and calls the fallback function of the Attacker;
3) The fallback function recursively calls the withdraw function again, i.e., *Reentrancy*;
4) Within an iteration bound, extra ether will be transferred multiple times to the Attacker.

V. Proposed Framework

As the example in Figure 1 demonstrates, there are two parts to check for vulnerability. First, a call to an external function is executed. Second, there is a write to a persistent state variable after the external call. Therefore, the pattern in the proposed analysis solution should combine both static and dynamic analysis methodologies to look for a persistent account state update after an external call to an untrusted address statically and to check if the recursive calling of such a function is possible dynamically. Our framework combines static and dynamic analysis to better capture the above-described attack scenario. The first stage of our framework uses static analysis and automated instrumentation of the smart contract under test and to efficiently and accurately capture the possible attack scenarios.

The second stage uses the information obtained from stage one to automatically generate an attacker contract which will be used in a later stage to interact with a deployed version of

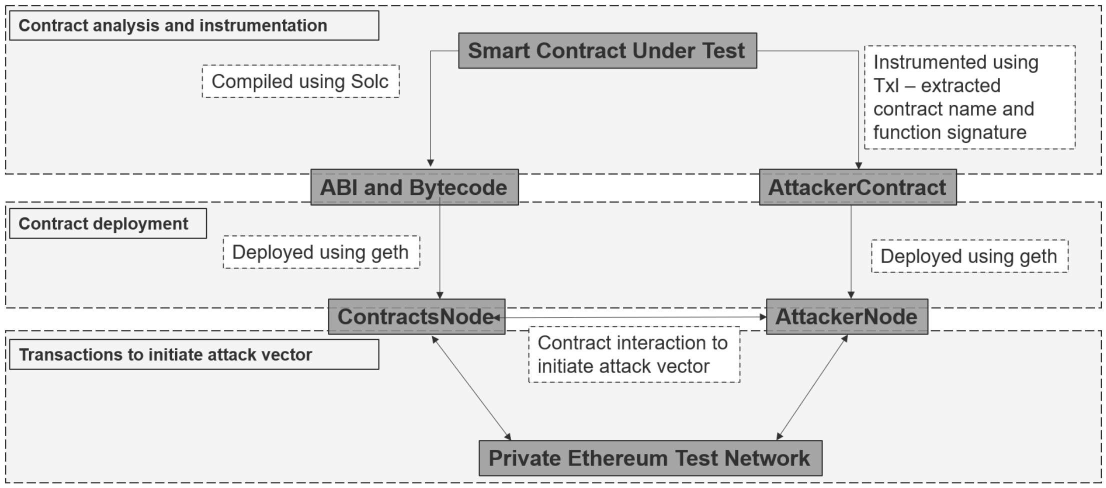

Contract analysis and instrumentation
Smart Contract Under Test
Compiled using Solc
Instrumented using Txl – extracted contract name and function signature
ABI and Bytecode
AttackerContract
Contract deployment
Deployed using geth
Deployed using geth
ContractsNode
AttackerNode
Transactions to initiate attack vector
Contract interaction to initiate attack vector
Private Ethereum Test Network

Fig. 3. Proposed Framework

the smart contact under-test and that do demonstrate dynamically whether the smart contract under-test has a confirmed *Reentrancy* vulnerability.

A. *Contract Analysis and Instrumentation*

To overcome the challenges in detecting *Reentrancy* vulnerability in Smart-Contracts, the proposed framework, presented in Figure 3, utilizes Txl [9], a programming language used for structural analysis and source transformation. The TXL paradigm (see Figure 2) consists of a grammar and transformation rules that are used to parse the input text into Abstract Syntax Tree (AST), and this intermediate AST is transformed into a new AST of the target domain, and finally, this output is unparsed to a new output text.

In our framework, we used TXL to automate the following tasks:

1) To parse the smart contract under test to identify potentially vulnerable functions, and to extract their names to help generate attacker contracts. Those contracts were aimed to be used in a later stage to create a *Reentrancy* attack scenario.
2) To instrument the contract and that to analyze the runtime behaviour of the smart contract under test during the attack.

However, the existing TXL grammar resources to be used with TXL did not have one for Solidity programming language, this called for the creation of a TXL grammar for Solidity [6] to enable parsing of solidity smart contracts using TXL.

The potentially vulnerable functions were determined if they contained an external call using any of these three Solidity's built-in functions: *transfer()*, *call()*, *send()*. These function signatures were extracted using TXL and outputted in a .txt file. This output file was used to modify the Attacker Contract to exploit *Reentrancy* vulnerability in the smart contract under test if it existed.

An *AttackerContract*, as shown in Listing5, is automatically created to interact with potentially vulnerable functions of the smart contract under test with a *Reentrancy* attack scenario.

For illustration, *FairDare* smart contract is used as an example from the collected dataset for this research. This framework successfully detected *Reentrancy* vulnerability within it in our experiment.

As shown in Listing5, the *AttackerContract* tries to request ether multiple times from the smart contract under test with *Reentrant* attack. At first, the AttackerContract creates an instance of the *FairDare* contract and thus calls the constructor of the *FairDare* contract. It then calls the withdraw() function of *FairDare* smart contract to initiate the attack (see Listing 6). Within the withdraw() function, the *FairDare* smart contract sends ether with a transfer() before setting the value of the corresponding *depositAmount* to 0. Since the call has no parameters provided, it will invoke the callback function of the *AttackerContrac*t.

```
import "FairDare.sol";
contract AttackerContract_FairDare{
    address payable private _owner;
    address payable private _vulnerableAddr;
    FairDare public fd =FairDare(_vulnerableAddr);
    constructor() public {
         _owner = msg.sender;
    }
    function() external{
        fd. Withdraw();
    }
    function transferToOwner() public {
        _owner.transfer(address(this).balance);
    }
}
```

Listing 5. AttackerContract

Within the callback function, the *AttackerContract* can invoke the withdraw() function again as the *Reentrancy* call. As a result, the *FairDare* smart contract will send ether to the *AttackerContract* again until all its ether is depleted. With the help of the *AttackerContract*, we can verify if the potential *Reentrancy* vulnerability detected by TXL pattern matching is exploitable.

```
function withdraw() public {
    require(tx.origin == msg.sender, "");
    uint blocksPast = block.number - depositBlock[
        msg.sender];
    if (blocksPast <= 100) {
        uint amountToWithdraw = depositAmount[msg.
            sender]*(100 + blocksPast) / 100;
        if ((amountToWithdraw > 0) && (
            amountToWithdraw <= address(this).
            balance)){
            msg.sender.transfer(amountToWithdraw);
                depositAmount[msg.sender] = 0;
        }
    }
}
```

Listing 6. Withdraw() function of FairDare Smart Contract (Smart contract under test) [10]

B. Contracts Deployment

For the contract deployment stage, we need to choose an Ethereum Private Test Network Specifications. In this framework, we use Geth [11], a Golang implementation of Ethereum, to create a private blockchain. This enabled us to create a new, and private blockchain from scratch which we will be using to deploy and test any smart contract. This private blockchain will not be connected to the Ethereum main net and therefore does not require real ether. To use Geth [11], we created a new account that represents a key pair. We simulated having multiple computers storing our blockchain by creating two nodes, which act like two different computers hosting and interacting with the same blockchain. Each node that wants to interact with a blockchain will store the blockchain on their computer. We initiated these two nodes from Geth using two different Terminal windows to simulate two different computers. We used two different data directories, so each node has a separate place to store their local copy of our blockchain as shown in the below Genesis Block,

```
{
 "config":
    {
        "chainID": 1708,
        "homesteadBlock": 0,
        "eip150Block": 0,
        "eip155Block": 0,
        "eip158Block": 0
    },
 "alloc":
    {
      "0xB1C0a62c5df3AE6469031D5BC0842382187C7F25":
        {
        "balance": "1000000000000000000000000000000"
        }
    },
 "difficulty": "0x4000",
 "gasLimit": "0xffffffff",
 "nonce": "0x0000000000000000",
}
```

Listing 7. CustomGenesis.json file

Our CustomGenesis.json (See Listing 7) file determines:

1) Contents of the genesis block, or the first block of our blockchain.
2) Configuration rules that our blockchain will adhere to.
3) homesteadBlock: defines the version of the Ethereum platform, we set this attribute to 0 as we are already on the version of Homestead version.
4) eip150Block/eip155Block/eip158Block: Ethereum Improvement Proposals (EIPs) indicate hard-forking for backward-incompatible protocol changes, we set this attribute to 0 as our private network will not require this hard-forking.
5) difficulty: On our test network, we kept this value low to avoid waiting during tests, since the generation of a valid Block is required to execute a transaction on the Blockchain.
6) gasLimit: Gas is Ethereum's fuel that is spent during transactions. We set this value high enough in our test network to avoid being limited during tests.
7) alloc: We used this attribute to create our wallet and prefill it with fake ether.

C. Interacting with Blockchain

To interact with the blockchain, we launched the geth *console* command to initiate the ports for ContractsNode and AttackerNode. After simulating the contracts node and attacker node on our private Ethereum network, the contracts under test are deployed on these nodes in the following steps:

1) Compiling the contract under test using either "solcjs" command or in "remix online IDE" [12]. After compiling the contract, the ABI and bytecode of the contract under test are generated.

```
[
   {"constant": false,
   "inputs": [],
   "name": "withdraw",
   "outputs": [],
   "payable": false,
   "stateMutability": "nonpayable",
   "type": "function" [10]
   },
   {"payable": true,
    "stateMutability": "payable",
    "type": "fallback"
   }
]
```

Listing 8. ABI code of FairDare smart contract

2) Using the ABI and bytecode, the contract is deployed on the network.
3) After deploying the contract onto a node, the transaction is submitted to all the peers of the blockchain. The miner.start() command starts the mining process and miner.stop() ends this process. After the mining process is completed, the contract gets added to the blockchain. And, once added to the blockchain, this contract cannot be changed.

Since we set the *–nodiscover* flag while connecting to our blockchain, these nodes cannot automatically interact with other nodes. Therefore, to enable transactions, we had manually tell one of the nodes about the other node as follows using the *admin.nodeInfo.enode()* and *admin.addPeer()* commands.

TABLE I
FUNCTION NAME EXTRACTED FROM CONTRACTS UNDER TEST

Contract Name	Function Name
DeFi	withdraw()
Globalcryptox	constructor()
FairDare	withdraw()
Moneybox	withdraw()
AIRToken	burn()
Quiz$_B LZ$	try()

After connecting both the nodes, transactions can be sent from an account to another using:

eth.sendTransaction (from: eth.accounts[0], to: "", value:"")

A specific function of a contract can be called using,

contractAddress.call.value ("ether to be transferred") .gas ("amount of gas to be spent") (abi.encodeWithSignature ("functionName(types)" , parameter values));

Where, *ContractAddress* is the address generated when a contract is deployed, *abi.encodeWithSignature* is used to encode structured data to aid in building valid call to functions.

The function name was extracted using TXL and substituted in the above command and the *AttackerContract* as well. TableI lists the extracted function name for each contract under test.

The complete process of deploying contracts and sending transactions can be depicted as follows: After successfully initiating transaction between *FairDare* smart contract and the modified *AttackerContract* for *FairDare* smart contract, the *Reentrancy* vulnerability in the smart contract under test was exploited as the fallback function of AttackerContract called the withdraw () function of the *FairDare* smart contract recursively till insufficient funds exception was thrown and only the last call to withdraw () function was reverted. The ether balance was tracked to monitor the *Reentrancy* bug existence as the ether of *FairDare* smart contract gets depleted multiple times and ether balance at the attacker node increases more than what it had requested. (See Figure 5)

VI. RELATED WORK

di Angelo and Salzer [13] presented a state of the art survey on tools for Ethereum Smart Contracts analysis. For a detailed comparison of those tools, the reader is advised to review this survey [13]. In our paper, we only discuss two tools that are closest to our approach, Contract Fuzzer [14], and Reguard [4].

A. Contract Fuzzer

Contract Fuzzer is a tool developed by Jiang et al. [14] to test the Smart-Contracts for vulnerabilities using a fuzzing technique. To detect the vulnerabilities, this tool starts with an initial analysis of the interfaces that the Smart-Contract

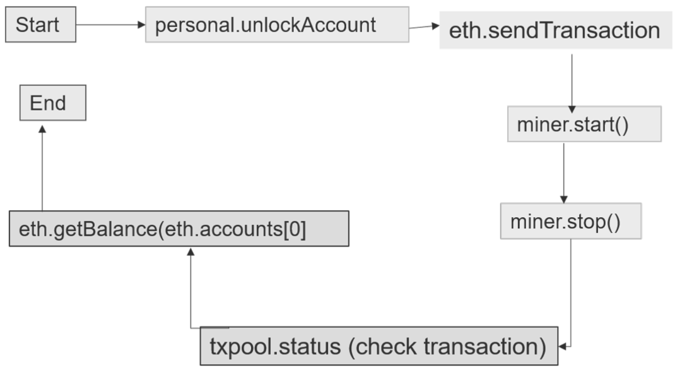


Fig. 4. Blockchain process of sending a transaction

exposes, it then randomly develops fuzzing inputs for these interfaces and observes the execution logs of the application. This tool targets more than one known vulnerabilities in Smart contracts, however, focusing on detecting *Reentrancy* vulnerabilities in smart contracts it may have slower performance compared to our proposed framework because it triggers test transactions against all the functions of a smart contract and not the potentially vulnerable functions by parsing the contract under test early on.

B. Reguard

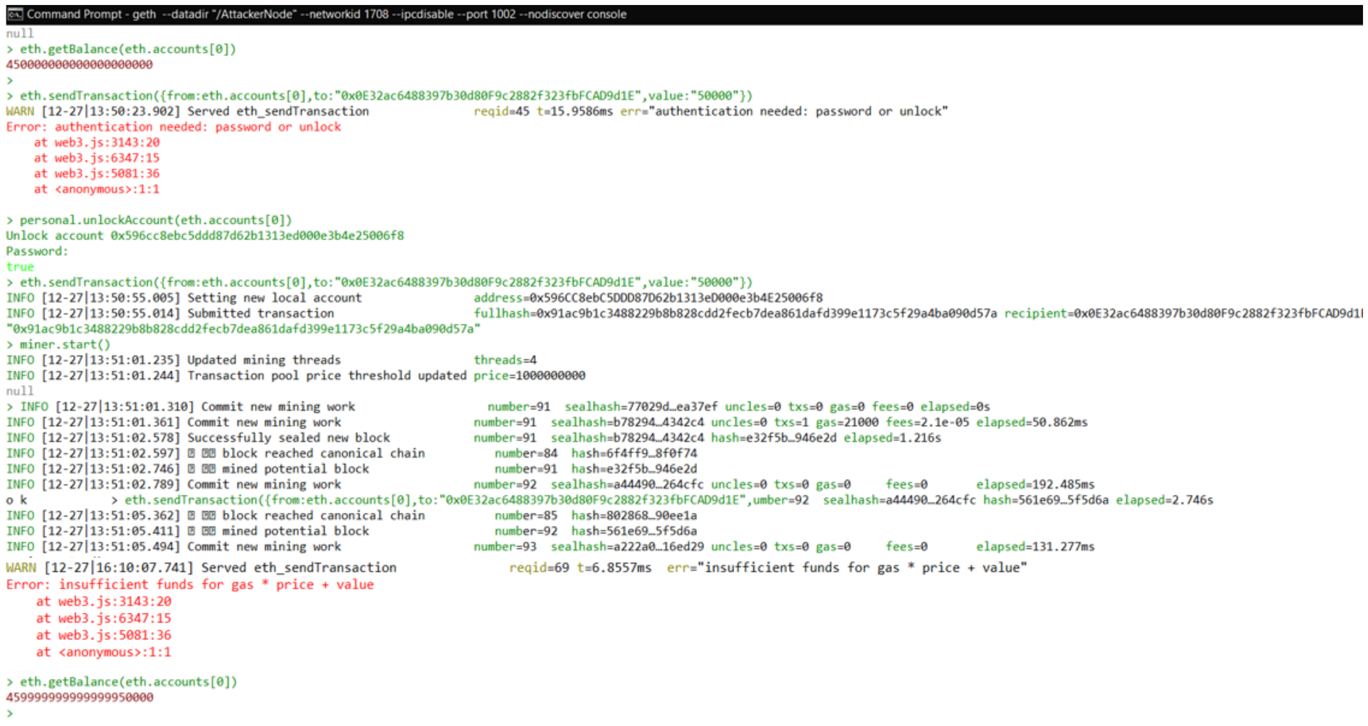
ReGuard is a dynamic analysis tool to detect *Reentrancy* vulnerabilities in Smart-Contracts developed by Liu et al. [4]. This tool tests the Smart-Contracts by initially transforming the Smart-Contract code into C++ and then generating fuzzing inputs to recreate Blockchain transactions as possible attacks. Then, ReGuard performs vulnerability detection through dynamic analysis. In comparison with this tool, we believe our framework will provide better performance as static analysis of smart contracts is done in its original form, i.e. using a context-free grammar for Solidity and not by transforming the smart contracts into some other programming language and subjecting the transformed contract to vulnerability test.

VII. EVALUATION

The dataset we used for the evaluation was extracted from Etherscan [10], a free-to-use platform for Blockchain analytics based on Ethereum. It is essentially a Block Explorer that

TABLE II
DETAILS OF 6 MODIFIED CONTRACTS UNDER TEST

Contract Name	Compiler Version	Balance	LOC
DeFi	V0.5.12	0.036 ether	449
Globalcryptox	V0.4.25	0 ether	235
FairDare	V0.5.12	0.0071 ether	41
Moneybox	v0.5.13	0 ether	40
AIRToken	V0.5.13	0 ether	304
Quiz$_B LZ$	v0.5.12	0 ether	54



```
Command Prompt - geth  --datadir "/AttackerNode" --networkid 1708 --ipcdisable --port 1002 --nodiscover console
null
> eth.getBalance(eth.accounts[0])
45000000000000000000
>
> eth.sendTransaction({from:eth.accounts[0],to:"0x0E32ac6488397b30d80F9c2882f323fbFCAD9d1E",value:"50000"})
WARN [12-27|13:50:23.902] Served eth_sendTransaction              reqid=45 t=15.9586ms err="authentication needed: password or unlock"
Error: authentication needed: password or unlock
    at web3.js:3143:20
    at web3.js:6347:15
    at web3.js:5081:36
    at <anonymous>:1:1

> personal.unlockAccount(eth.accounts[0])
Unlock account 0x596cc8ebc5ddd87d62b1313ed000e3b4e25006f8
Password:
true
> eth.sendTransaction({from:eth.accounts[0],to:"0x0E32ac6488397b30d80F9c2882f323fbFCAD9d1E",value:"50000"})
INFO [12-27|13:50:55.005] Setting new local account               address=0x596CC8ebC5DDD87D62b1313eD000e3b4E25006f8
INFO [12-27|13:50:55.014] Submitted transaction                   fullhash=0x91ac9b1c3488229b8b828cdd2fecb7dea861dafd399e1173c5f29a4ba090d57a recipient=0x0E32ac6488397b30d80F9c2882f323fbFCAD9d1
"0x91ac9b1c3488229b8b828cdd2fecb7dea861dafd399e1173c5f29a4ba090d57a"
> miner.start()
INFO [12-27|13:51:01.235] Updated mining threads                  threads=4
INFO [12-27|13:51:01.244] Transaction pool price threshold updated price=1000000000
null
> INFO [12-27|13:51:01.310] Commit new mining work                  number=91  sealhash=77029d…ea37ef uncles=0 txs=0 gas=0 fees=0 elapsed=0s
INFO [12-27|13:51:01.361] Commit new mining work                  number=91  sealhash=b78294…4342c4 uncles=0 txs=1 gas=21000 fees=2.1e-05 elapsed=50.862ms
INFO [12-27|13:51:02.578] Successfully sealed new block           number=91  sealhash=b78294…4342c4 hash=e32f5b…946e2d elapsed=1.216s
INFO [12-27|13:51:02.597] □ □□ block reached canonical chain        number=84  hash=6f4ff9…8f0f74
INFO [12-27|13:51:02.746] □ □□ mined potential block                number=91  hash=e32f5b…946e2d
INFO [12-27|13:51:02.789] Commit new mining work                  number=92  sealhash=a44490…264cfc uncles=0 txs=0 gas=0     fees=0       elapsed=192.485ms
o k           > eth.sendTransaction({from:eth.accounts[0],to:"0x0E32ac6488397b30d80F9c2882f323fbFCAD9d1E",umber=92  sealhash=a44490…264cfc hash=561e69…5f5d6a elapsed=2.746s
INFO [12-27|13:51:05.362] □ □□ block reached canonical chain        number=85  hash=802868…90ee1a
INFO [12-27|13:51:05.411] □ □□ mined potential block                number=92  hash=561e69…5f5d6a
INFO [12-27|13:51:05.494] Commit new mining work                  number=93  sealhash=a222a0…16ed29 uncles=0 txs=0 gas=0     fees=0       elapsed=131.277ms
WARN [12-27|16:10:07.741] Served eth_sendTransaction                 reqid=69 t=6.8557ms  err="insufficient funds for gas * price + value"
Error: insufficient funds for gas * price + value
    at web3.js:3143:20
    at web3.js:6347:15
    at web3.js:5081:36
    at <anonymous>:1:1

> eth.getBalance(eth.accounts[0])
45999999999999950000
>
```

Fig. 5. Ether balance after submitting transaction and after mining

allows users to easily lookup, confirm and validate transactions that have taken place on the Ethereum Blockchain. Smart-Contracts developers can get a substantial benefit from APIs available that can be utilized to either build decentralized applications or serve as a dataset for Ethereum Blockchain analysis.

We modified 6 contracts from Etherscan to test for *Reentrancy* vulnerability. The modification was done to introduce *Reentrancy* vulnerability in the selected contracts by updating the token balance after transferring the ether to an external contract. Table II shows the specifications of the contracts used.

We have conducted a preliminary evaluation of our proposed framework on 6 modified smart contracts from Etherscan [10] and our framework was able to detect the *Reentrancy* vulnerability in all of our modified contracts, with an exception of one contract that was caused due to version incompatibility of the context-free grammar created for Txl parser. We are evolving our TXL grammar to account for the various versions of Solidity.

We used Reguard [4] and ContractFuzzer [14] to compare and to evaluate the accuracy of the proposed framework. While ContractFuzzer allows checking on a set of predefined vulnerabilities patterns in smart contracts, we focused on *Reentrancy* vulnerability in this evaluation. As ContractFuzzer targets every function of a contract under test, it covers a wider attack surface but produced more false positives compared to our framework which included analyzing a contract statically and targeting potentially vulnerable functions only which lead to reduced false positives (See Table III). We were not able to access the Reguard tool to analyze our dataset but we believe our proposed framework may return better results in performance than Reguard because, in our framework, Smart contracts are directly analysed and not transformed into a C++ intermediate representation.

TABLE III
RESULTS OF *Reentrancy* VULNERABILITY ANALYSIS. FP: FALSE POSITIVE. FN: FALSE NEGATIVE. * : ANALYSIS FAILED

Contract Name	ContractFuzzer	OurFramework
DeFi	(1-FP)2	1
Globalcryptox	1	*
Moneybox	2	2
AIRToken	(2-FP) 3	1
FairDare	1	1
Quiz$_{B}LZ$	*	1

VIII. CONCLUSION

In this paper, a combined static and dynamic analyzer framework is proposed to detect *Reentrancy* vulnerabilities in Ethereum smart contracts. This framework leverages TXL for analyzing these smart contracts to extract the function signature of potentially vulnerable functions in a contract. Then, the *Reentrancy* bug detection is done at the run-time while interacting with the vulnerable contract via an attacker contract and that by trying to recreate the *Reentrancy* scenario.

IX. Future Work

The future work in this research would be to deploy our approach as a tool and that by automatically integrating the three automated stages of our approach (Analysis, Deploy and Test) of smart contracts for *Reentrancy* vulnerability. This automated tool would also be scalable to include Solidity smart contracts of newer versions. Currently, this framework targets Solidity smart contracts of versions after v4.28. The Dataset used in this work utilizes 6 modified contracts from Etherscan [8]. However, this can be increased to test many more smart contracts by developing a web scraper and running on the Etherscan portal to collect many smart contracts deployed on this website.

Acknowledgments

This work is supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] "Bitcoin home page," last accessed on 01-10-2019. [Online]. Available: https://bitcoin.org/

[2] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts sok," in *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. New York, NY, USA: Springer-Verlag New York, Inc., 2017, pp. 164–186.

[3] A. Dika and M. Nowostawski, "Security vulnerabilities in ethereum smart contracts," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, July 2018, pp. 955–962.

[4] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts." in *ICSE (Companion Volume)*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 65–68.

[5] "Ethereum home page," last accessed on 01-10-2019. [Online]. Available: https://www.ethereum.org/

[6] "Solidity home page," last accessed on 01-10-2019. [Online]. Available: https://solidity.readthedocs.io/en/v0.5.1/

[7] H. Hasanova, U.-j. Baek, M.-g. Shin, K. Cho, and M.-S. Kim, "A survey on blockchain cybersecurity vulnerabilities and possible countermeasures," *International Journal of Network Management*, vol. 29, no. 2, p. e2060, 2019, e2060 NEM-18-0162.R1.

[8] M. Saad, J. Spaulding, L. Njilla, C. A. Kamhoua, S. Shetty, D. Nyang, and A. Mohaisen, "Exploring the attack surface of blockchain: A systematic overview," *CoRR*, vol. abs/1904.03487, 2019. [Online]. Available: http://arxiv.org/abs/1904.03487

[9] "Txl home page," last accessed on 01-10-2019. [Online]. Available: http://txl.ca/

[10] "Etherscan home page," last accessed on 01-09-2019. [Online]. Available: https://etherscan.io/

[11] "Geth home page," last accessed on 01-10-2019. [Online]. Available: https://geth.ethereum.org/downloads/

[12] "Remix home page," last accessed on 01-11-2019. [Online]. Available: https://remix.ethereum.org/

[13] J. Gao, H. Liu, Y. Li, C. Liu, Z. Yang, Q. Li, Z. Guan, and Z. Chen, "Towards automated testing of blockchain-based decentralized applications," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, May 2019, pp. 294–299.

[14] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 259–269.

[15] 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)*, April 2019, pp. 69–78.

[16] 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: ACM, 2016, pp. 254–269. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978309

[17] M. Demir, M. Alalfi, O. Turetken, and A. Ferworn, "Security smells in smart contracts," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2019, pp. 442–449.

[18] Z. Wan, D. Lo, X. Xia, and L. Cai, "Bug characteristics in blockchain systems: A large-scale empirical study," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, May 2017, pp. 413–424.