Ethereum Smart Contract Security Analysis

Kazi Alom, Lay Jain, Srijon Mukherjee

The Ethereum blockchain offers smart contracts as a way of handling financial promises. These are implemented as Turing-complete programs allowing a smart contract creator to execute arbitrary code in handling transactions. These smart contracts are available for public viewing and are susceptible to being exploited. We aim to do a security analysis on common smart contracts and explore what kinds of critical vulnerabilities may exist.

***Index Terms*—security analysis, ethereum, smart contract**

I. Foundations

Blockchain technology's greatest strength is allowing the decentralization of historically centralized services. Bitcoin [8] has a market cap of almost \$200 billion and sees a daily trading volume of about \$100 billion. Besides Bitcoin, there are countless other "alt-coins" that each have their own platform and offer a type of service. Ethereum [1] is one platform that allows digital assets be directly controlled by a piece of code called a smart contract. Smart contracts essentially allow systems that automatically move digital assets according to pre-defined rules. Decentralized autonomous organizations (DAOs) are long-term smart contracts that are controlled by shareholders. The DAO [7], a crowdfunded venture capital fund that existed as a smart contract, had over \$50 million stolen due to an exploited bug in the smart contract language Solidity. In fact, this incident is the reason why Ethereum was hard-forked and why we see another alt coin "Ethereum Classic". Other high profile smart contracts have also been targeted and lost their funds to these types of attacks. As the usage of smart contracts becomes more prevalent, it is critical to address bugs in smart contract implementations and in Solidity itself.

Smart contract vulnerabilities can be classified [13] as blockchain vulnerabilities, Solidity vulnerabilities, and software security vulnerabilities. The Transaction Ordering Dependency problem is one blockchain vulnerability which involves a new block on the chain containing multiple transactions invoking the same contract. There is no certainty in the state of the contract when either individual transaction invokes the contract because the order is not known until after the block is mined. The Timestamp dependency problem says that a smart contract using timestamps of blocks can potentially be manipulated by malicious miners who modify their local system's time.

Solidity compiler vulnerabilities are bugs within the smart contract's high level language and how it generates the Ethereum Virtual Machine (EVM) bytecode. There have not been many CVE's published on this which makes the EVM an excellent candidate for fuzzing.

Software security vulnerabilities are those due to erroneus smart contract code and improper Solidity practice. The DAO attack was victim to a re-entrancing attack where the contract uses an unsafe function call.value() that allows the attacker to take advantage of a callback to recursively withdraw funds from a victim contract. Other usual security vulnerabilities such as buffer overruns and integer over/underflows also fall under software bugs.

The parity multisig attack was one incident where \$50 million of Ethereum was frozen due to an attacker calling an unprotected function and gaining ownership of a public smart contract library that was imported by other contracts. The attacker invoked the kill function that removed the contract from the blockchain. This could have been mitigated by using a private modifier on the function that allowed the attacker to escalate privileges.

Ethereum smart contracts rely on "gas", which is additional transaction fee paid to the miner for borrowing their computation ability on the blockchain. If this gas runs out before an operation is over, the callee contract throws an exception and these must be properly checked by the caller contract. This applys to all other exceptions that can be raised such as when exceeding stack capacity, or any other unknown system error occurs. These also fall under software security vulnerabilities that can be mitigated by safe code practice.

II. Common Vulnerabilities

As mentioned, there exist a variety of smart contract weaknesses [15] due to software errors, EVM exceptions, blockchain errors, etc. It helps to look at some Solidity code examples and understand how exactly the exploits would be triggered. The following samples demonstrate vulnerabilities besides crypto implementation flaws, obsolete function usages, and typical programming malpractice.

Public access to critical private data: When a contract uses the `private` modifier on a function or field, it does not mean that these variables cannot be read. Any attacker can look at the transactions related to this contract on the public blockchain to figure out the state of all variables. In the following example, an attacker can simply calculate what number they need to trigger `selectWinner` into transferring the contract balance to his address by first observing the value at `numbers[0]`. This would be seen the arguments to `play` in a previous transaction. One way to resolve this for a "game" like the contract below is to use a commitment scheme that hides values using hashes until both players have picked numbers.

```
contract Test{
 uint[2] private numbers;
 address[2] private addrs;
 uint idx = 0;

 function play(uint number) public payable {
     require(msg.value == 1 ether,
         'must be called with 1 ether');
     numbers[idx] = number;
     addrs[idx] = msg.sender;
     idx++;
     if (idx == 2) selectWinner();
 }

 function selectWinner() private {
     uint n = (numbers[0] * numbers[1]) % 2;
     (bool success,) =
      addrs[n].call.value(address(this).balance);
     require(success, "transfer failed");
     delete numbers;
     delete addrs;
     idx = 0;
 }
}
```

DoS by uncontrolled gas consumption: All smart contracts consume gas depending on how many instructions must be computed based on what function was called. This gas price and amount is specified in the transaction sent to the contract. Miners are incentivized by higher gas prices thus transactions with high gas prices are usually completed quicker. In Ethereum, the sum of all transactions in a block cannot exceed a gas limit threshold. If a contract like the one below maintains an array of unbound size, it can lead to a denial-of-service condition where the function that loops across the array's values may exceed the block gas limit. This means that it is critical that developers do not loop over arrays that are expected to grow over time.

```
contract Test {
    address[] addr_list;

    function addAddress() public {
        addr_list.push(msg.sender);
    }

    function transferAddresses() public payable {
        for(uint i=0; i < addr_list.length; i++) {
            // doing ANYTHING is a DOS risk
        }
    }
}
```

Arbitrary data write: Similar to how in a standard buffer overflow an attacker can overwrite critical data such as a function return address, there exists a similar problem where anyone may overwrite the owner of a contract. This would then allow them to call privileged functions such as a withdrawal that validates the sender address of a transaction against the contract owner. Developers should ensure that there are no out-of-bounds accesses where writes to one data structure might corrupt another data structure in the address space.

```
contract Test {
    address public owner;
    uint256[] map;

    function Test() public {
        owner = msg.sender;
    }

    function set(uint256 key, uint256 value) public {
        map[key] = value;
    }

    function get(uint256 key)
        public view returns (uint256) {
        return map[key];
    }

    function withdraw() public {
        require(msg.sender == owner);
        msg.sender.transfer(address(this).balance);
    }
}
```

Insufficient entropy for random values: One popular use of smart contracts may be in gambling applications, which typically rely on pseudorandom number generators to pick winners. Due to the state of a contract being public on the blockchain, having a source of randomness is non-trivial. For instance, relying on the timestamp as a random value does not work because a miner can arbitrarily set their local machine times. One way to solve this is to use a commitment scheme to commit a guess and answer before revealing.

```
contract Test {
  uint8 answer;

  function init_challenge() public payable {
      require(msg.value == 1 ether);
      // hash(previous blocks hash | timestamp)
      answer = uint8(keccak256(
          block.blockhash(block.number - 1), now));
  }

  function guess(uint8 g) public payable{
      require(msg.value == 1 ether);

      if (g == answer) {
          msg.sender.transfer(address(this).balance);
      }
  }
}
```

Race condition: By nature of being on a blockchain, some transactions that call the same contract may occur in the same block and thus their ordering depends on the miner who completes it. One simple example is if there was a contract that awarded tokens to the first person to guess a correct value. Let's say you figured out the answer and send your transaction onto the network. In the time it takes for the block to get mined, an adversary sees your transaction and copies the answer you found and sends their own transaction, except they provide a much higher gas value. Rational miners would complete the adversary's transaction before yours, letting the adversary win the money instead. To mitigate this situation, the contract can store the salted hash of the answer and then compare it against the salted hash of your guess.

```
contract Test {
    address public owner;
    uint public prize;
    bool finished;

    function Test() public {
        owner = msg.sender;
        finished = false;
    }

    function setPrize() public payable {
        require(!finished);
        require(msg.sender == owner);
        owner.transfer(prize);
        prize = msg.value;
    }

    function claimPrize(uint guess) {
        require(!finished);
        require(guess == keccak256("secret"));
        msg.sender.transfer(prize);
        finished = true;
    }
}
```

DoS through improper exception handling: Contracts that perform external calls such as sending a payment to another address may be susceptible to a denial-of-service condition if this call occurs in a loop. In other words, it is bad to attempt multiple payments in one transaction because if one of them fails, the rest of the payments will not happen and the funds are effectively frozen until the exception somehow does not occur anymore.

```
contract Test {
 address[] refundAddresses;
 mapping (address => uint) public refunds;

 function Test() {
  refundAddresses.push(0xdeadbeef);
  refundAddresses.push(0xd15ea5e);
 }

 function refundAll() public {
  for (uint x; x < refundAddresses.length; x++) {
  // exception on x=0 causes refundAddresses[1]
  // to never get a refund
      require(refundAddresses[x].send(
         refunds[refundAddresses[x]]));
  }
 }
}
```

Integer under/overflow: This type of vulnerability happens when an arithmetic operation results in a value greater than the maximum or smaller than the minimum for a specific type. For instance, trying to store 2^8 in a `uint8` type would actually store a 0 because the value wraps around. This is mitigated by using a safe math library that checks for overflowing

```
contract Test {
    uint public number = 1;

    function sub(uint guess) public {
        number -= guess;
    }
}
```

Unprotected self-destruct: This is a simple improper access control vulnerabilitiy but it is significant because it is the reason why the parity multisig attack occurred. An attacker was able to escalate privileges and self-destruct a library imported by many other contracts freezing all the funds in those contracts. In order to defend against this vulnerability, a developer should implement proper access controls if needing this functionality at all.

```
contract Test {
    function suicide() {
        selfdestruct(msg.sender);
    }
}
```

Re-entrancy: This is a relatively common and devastating attack where calling an external contract allows an attacker to take over control flow by recursively calling back into the contract before the initial call's appropriate state changes occur. The DAO attack was victim to re-entrancy where an attacker recursively withdrew funds using a single call. Similarly, in the example below, the condition that allows a caller to trigger a transfer depends on a state that is not changed until after the external call occurs. The `msg.sender.call.value(amount)` allows an attacker to specify a function they can call back into after the transfer happens but before the necessary state change. This vulnerability is mitigated by performing all of a contract's state changes before doing an external call.

```
contract Test {
    mapping (address => uint) public credit;

    function donate(address to) payable public {
        credit[to] += msg.value;
    }

    function withdraw(uint amount) public {
        if (credit[msg.sender] >= amount) {
            require(msg.sender.call.value(amount)());
            // this state change occurs after call
            credit[msg.sender] -= amount;
        }
    }
}
```

III. Initial Approach

The plan was to use Manticore [9], a symbolic execution engine that supports EVM bytecode (and other architectures) to maximize code coverage on smart contracts sampled from Etherscan [4]. Manticore can simulate the EVM as well as simulate an entire ”Ethereum world” where a contract may call on other contracts. One advantage is that this analysis can be done without any real Ethereum coin because Manticore can set arbitrary amounts for contracts and wallets you define in the API.

The objective was to identify devastating vulnerabilities we have seen before such as the re-entrancing attack on the DAO that allows an attacker to steal funds. Manticore would provide inputs that will hit various branches in smart contracts and the next step is to apply a fuzzer to user-controlled inputs in order to generate corner cases. In other words, we want to cause as many exceptions we can along every branch in the code and log them for further analysis. The collected smart contracts will also be analyzed for semantic bugs after reading up on the latest version of Solidity and recommended practice.

One difficulty in the approach may be dealing with all the versions of Solidity being used in the wild. Solidity is constantly being patched which causes smart contract writers to be slow in keeping up. Manticore fully supports Solidity versions pre-0.5.0 so we will have to scrape contracts using Solidity 0.4 and older.

Once a vulnerabilitiy is found, it will be tested on an online Ethereum development environment called Remix [14]. Remix makes it simple to write Solidity code, compile it into bytecode, and interact with the Ethereum test networks. The user can also set a blockchain state to force the contract to execute under arbitrary parameters.

IV. Approach

After some more reconnaisannce, it turned out that most contracts on Etherscan are actually honeypot contracts. Honeypot contracts look vulnerable but actually have some hidden state that is difficult to discern at a first glance, and can thus trick malicious actors into sending it money. We take a look at some of these, but we also expand our scope to contracts that are not shown on Etherscan.

Etherscan shows the last five hundred verified contracts on the blockchain, meaning that the owner publicly published what they claim is the source. Given that the blockchain is always moving with nearly a million transactions a day, there are vastly more contracts than the ones shown on Etherscan. The issue is that Manticore does not support execution on bytecode. Most of the contracts on the blockchain do not have verified source code and are only visible as the EVM bytecode they were compiled as. Moreover, there is little incentive to analyze verified source code contracts if most of them are honeypots. We used a tool Pakala [12] that does symbolic execution on EVM bytecode instead of Manticore to look for vulnerable contracts. This method allowed us to perform mass scanning of bytecode so we could try to hit as many contracts as possible for common vulnerabilities. Another tool we explored is called Echidna [2] which does fuzzing of smart contracts in order to violate assertions made by a user in order to see how one might test their smart contract before deploying it.

V. Honeypot Contracts

VI. Mass Scanning with Pakala

To prepare for the mass scanning of smart contracts on the blockchain, we have to extract all the contract bytecodes. One way to do this would have been to run a local Ethereum node and wait for it to synchronize with the rest of the mining network. This would have meant waiting for it to download nearly 60 GB of data and then parse through all of it. Fortunately, there are public Google Cloud datasets for many popular cryptocurrencies including Ethereum. We decided to extract and scan contracts that have non zero balances using the following SQL query:

```
Select A.*, B.eth_balance
from 'bigquery-public-data.crypto_ethereum.contracts'
A inner join
'bigquery-public-data.crypto_ethereum.balances'
B on A.address = B.address where B.eth_balance > 0
```

This query returned 473594 smart contracts with non-zero balance which ended up being about 2 GB worth of JSON.

One complication with using Pakala is that it requires a URL to an Ethereum node so it can perform necessary RPC functions for retrieving the state of a contract on the blockchain. Initially, this was handled by running the Parity [10] mining software and using its local URL for Pakala. This was unsustainable because the machines running the scanners would run out of disk space trying to download the entire blockchain. After switching over to a third party node Infura [6], the scanners were able to call the RPCs over the internet to the remote nodes. The downside was that Infura limits free accounts to under 100000 requests a day.

The next step was to set up a python script that can spawn instances of Pakala to scan a contract. By running the command `echo "0xdeadbeef" | pakala - --exec-time 30 --analysis-time 30 --max-transaction-depth 5 -z`, we can feed in arbitrary EVM bytecode through `stdin` and specify timeouts in seconds for the analysis and fuzzing. We limit the transaction depth to 5 so that Pakala does not consider any complicated changes to the contract's state. Sending multiple transactions involves setting up the contract's long-term memory a certain way to execute an attack. The `-z` flag disabled concretization of symbolic values which helped to cut back the analysis time. The goal is to get through as many contracts as possible in the limited time available, so we enforce these strict parameters so the scanner does not hang on any one contract for too long.

The JSON of bytecodes was split into 6 parts and the scanner was run in 6 different processes on multiple computers in order to make a dent in the massive amount of contracts we scraped. The scanner was simply a python script that opened new processes for each bytecode in the JSON. When a Pakala process terminates, everything that was sent to its `stdout` gets appended as a new line to an output file. Each line of the output is marked with its offset into the JSON for convenient lookup later. To locate an identified bug, all you would have to do is open one of the output files for a scanner, and do a search for the string `"Bug"` because Pakala prints ”Bug Detected!” on discovering a vulnerability. The following shows the code for a single scanner feeding on one of the JSON files using python subprocesses to fork instances of Pakala.

```
import json
from subprocess import Popen, PIPE, STDOUT
import datetime

FILE_PATH = "./data-000000000001"

counter = 0 # what index to start at

out = open(f"output{FILE_PATH[2:]}.txt","a")

with open(FILE_PATH, "r") as fp:
  lines = fp.readlines()

for i in range(counter, len(lines), 1):
  out.write(f"\nCOUNTER␣{counter}")

  p = Popen(['pakala', '-', '--exec-time','30',
'--analysis-time', '30',
'--max-transaction-depth', '5',
'-z'],
      stdout=PIPE, stdin=PIPE, stderr=STDOUT)

  p_stdout = p.communicate(
input=bytes(json.loads(lines[i])['bytecode'][2:],
    "UTF-8"))

  out.write(str(p_stdout[0]))
  counter+=1

out.close()
```

After running for 24 hours, the scanners burned through 10887 contracts and detected 31 that were vulnerable. One vulnerable contract at the address `0x70025b7a4eC0baF3aE48352FAe41c81B62ee992E` with a balance of 0.51778 Ethereum was detected to have a self destruct bug. The Pakala output shows:

```
Symbolic execution finished with coverage 100%.
Outcomes: 84 interesting. 346 total and 0
 unfinished paths.

Starting analysis step...
Loaded 10 storage slots from the contract
 (non-exhaustive). 0 non-zero.

Found selfdestruct bug.
Path:

Transaction 1, symbolic state:
{
"selfdestruct_to": None,
"calls": [[<BV256 0x20>,
  <BV256 0x60>,
  <BV256 0x24>,
  <BV256 0x60>,
  <BV256 0x0>,
  <BV256 0xffffffffffffffffffffffffffffffffffffffff
    & calldata[0]_608_256[223:0] ..
     calldata[32]_609_32>,
  <BV256 gas_601_256 - 0x61da>]],
"storage_written": {<BV256 0x1>:
    <BV256 calldata[36]_610_256>,
    <BV256 0x0>: <BV256 calldata[0]_608_256[223:0] ..
     calldata[32]_609_32 &
     storage[<BV256 0x0>]_617_256>},
"storage_read": {<BV256 0x0>:
    <BV256 storage[<BV256 0x0>]_617_256>},
"env": {'balance': <BV256 0x128dfa6a90b28000>,
 'caller':
    <BV256 0xcafebabeffffffff0202ffffffff7cff7247c9>,
 'value': <BV256 value_600_256>},
"solver": {'constraints': [<Bool !calldata_size_604_256
   <Bool calldata[0]_608_256[255:224] == 0x7948f523>,
   <Bool storage[<BV256 0x0>]_617_256[159:0] == 0x0>,
   <Bool calldata[32]_609_32 == 0xff7247c9>,
   <Bool calldata[0]_608_256[127:0] == 0xcafebabeffffff
   <Bool CALL_RETURN[<BV256 0xfffffffffffffffff &
   calldata[0]_12_256[223:0] ..
   calldata[32]_14_32>]_26_256[159:0] ==
    0xdeadbeef00000000000000000000000000000000>],
 'hashes': {}},
}
```

Pakala works by identifying all branch instructions in the bytecode and generates symbolic expressions that satisfy the constraints of all indirect jumps. The point of this is to maximize coverage of the subject contract by hitting as many distinct paths of the control flow graph as feasible with the allotted timeouts. At each possible state of the contract, Pakala checks if it is possible for a caller's balance to be greater than it was before calling the contract. This invariant check being true implies that the caller was able to send an input to the contract that transferred funds into the caller's account.

As seen in the result above, Pakala was able to cover 100% of paths in this contract and found a self-destruct bug. This means that it found one path where it was able to call the self-destruct function on the contract which forwards all remaining funds to the caller before disabling the contract completely. Pakala uses the Z3 [3] SMT solver to obtain concrete values from the symbolic state that triggered this bug. In this case, the concrete state tells us what bytes the `calldata` data structure should contain at various indices in order to trigger this bug. The `storage_read` and `storage_written` fields tell us that the contract's blockchain state must also be set up in a particular way. We can experimentally verify this self-destruct exploit if we can send a transaction with this exact state to the contract

Unfortunately, Remix does not support interacting with bytecode contracts, so it is difficult to dynamically analyze what is going on at each instruction of the code. This makes it infeasible to attempt exploiting this vulnerability, or any of the other vulnerabilities reported by Pakala unless we blindly send transactions to the actual contracts on the Ethereum network. There exist EVM disassemblers for seeing what the instructions of a contract are, but there are no emulators or debuggers that allow a developer to step through instructions and monitor the program state. The closest thing to an emulator is the actual code that is run by the Ethereum mining software when executing a contract for a block. Parity's `evmbin` is a compiled Rust binary that implements the real EVM by taking bytecode and outputting all the instructions along with the EVM state at each program counter. This state includes the current gas limit, stack, storage, and local memory values. It may be possible to use this `evmbin` library as a base for writing an EVM emulator or debugger that could then allow us to construct the exploits previously mentioned.

VII. Fuzzing with Echidna

Fuzzing refers to the process of feeding random input data to a program in order to potentially induce undesirable behavior. Sometimes the inputs are adaptively chosen in order to maximize coverage. The existence of unexpected behavior indicates the presence of a bug—potentially an exploitable vulnerability. Echidna [2] is a fuzzing tool for Ethereum smart contracts that checks for the violation of assertions or user defined properties (indicated using a special prefix). It simulates random calls to various public functions in the contract from a partially configurable set of senders. The user may specify which functions to call and can also monitor the maximum gas usage by each function. In case a property is violated, a sequence of calls leading to the violation is outputted (Echidna tries to shorten the violating sequence if possible). Echidna can thus detect several kind of elementary bugs such as ones resulting from race conditions, integer overflows, or uncontrolled gas usage. However, fuzzing alone is ineffective against more subtle issues like re-entrancy which might only be detectable if fuzzing is integrated with runtime monitoring and/or static analysis as some other tools attempt to do.

However, since Echidna is primarily intended as a tool for testing contracts, it requires a basic understanding of the code in order to be utilized effectively. Few contracts include assertions in the source and thus suitable properties need to be devised and implemented. Although some generic techniques such as checking ranges of balances and measuring gas consumption usually apply, they alone are likely to catch a vanishingly small number of bugs. Hence, the attack needs to be tailored for each contract thereby making Echidna unsuitable for large-scale vulnerability detection. Therefore, for our purposes, we use the static analysis tool Slither [5] to identify contracts which might have detectable issues based on an analysis of source code in order to narrow down the search. Note that Slither is much more susceptible to false positives as compared to Echidna as the latter produces an actual sequence of calls in which the issue materializes. On the other hand, this narrowing down significantly increases the rate of false negatives. However, resource limitations compel us to follow this strategy.

Slither only works on versions 0.4.x of Solidity greatly limiting the number of contracts that can be scanned. Among contracts with non-zero balance, Slither identified only a handful with potential bugs that could be detectable using Echidna. Running Echidna on these as well as a few randomly picked contracts revealed no vulnerabilities despite several hours of effort. No critical invariants were found to be broken and no functions took up unusual amounts of gas. However, Slither alone is capable of detecting a much larger class of errors such as re-entrancy. For instance, Slither could have detected the vulnerability that led to the DAO hack [5]. This raises questions as to the practicality of using fuzzers just to rule out false positives.

VIII. Conclusion

Although we could not exploit any vulnerable contracts, we lay the groundwork for assessing new contracts through a variety of open source tools. We would have initially fallen for honeypot traps if we had been careless and used Manticore to find vulnerabilities in verified source code. Pakala worked to find trivial bugs and vulnerabilities in bytecode by maximizing coverage and fuzzing inputs, but applying its results usefully was difficult due to a lack of EVM debugging capability. Echidna is a more precise fuzzing tool than Pakala that uses information from source code to find most of the common vulnerabilities discussed, but both tools lack the ability to find re-entrancy bugs. Slither can find re-entrancy bugs, but it has a high false positive rate and does not give concrete inputs to trigger such bugs.

An important next step in the black box analysis of EVM bytecode is to develop a flexible debugger that can step through instructions and insert breakpoints where local memory and blockchain storage can be inspected. This would make it easy to universally analyze contracts the same as one would reverse engineer an x86 executable file.

Clearly there are novel techniques consistently being innovated to analyze contracts where some are advantageous over others in certain cases and vice versa. These tools can be leveraged by developers to test their smart contracts before initializing them on the blockchain, but in practice it is recommended to instead use a method called "formal verification" as in the tool Securify [16]. We could not analyze with Securify because it is a proprietary software. Furthermore, formal verification is a tedious and expensive process as it requires manually writing specifications for the contracts to be tested. The alternative to formal verification for writing secure smart contract code is to use open source [11] libraries of contracts thoroughly vetted by the community. For instance, you may copy from a library that provides safe math functions where integer overflows are prevented. Regardless, vulnerabilities in open source libraries would compromise all the contracts that inherit from them.

References

[1] V. Buterin. Ethereum white paper: A next generation smart contract & decentralized application platform. 2008. "[Online; accessed 5-April-2020]".

[2] Cryptic. Echidna. https://github.com/crytic/echidna, 2020. [Online; accessed 12-May-2020].

[3] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[4] Etherscan. Etherscan. https://etherscan.io/aboutus, 2015. [Online; accessed 6-April-2020].

[5] 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*, WETSEB ?19, page 8?15. IEEE Press, 2019.

[6] Infura. Infura api suite. https://infura.io/product, 2020. [Online; accessed 12-May-2020].

[7] A. Madeira. The dao, the hack, the soft fork and the hard fork. 2019. "[Online; accessed 5-April-2020]".

[8] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008. "[Online; accessed 5-April-2020]".

[9] T. of Bits. Manticore. https://github.com/trailofbits/manticore, 2019. [Online; accessed 6-April-2020].

[10] OpenEthereum. Parity. https://www.parity.io/ethereum/, 2020. [Online; accessed 12-May-2020].

[11] OpenZeppelin. Openzeppelin contracts. https://github.com/OpenZeppelin/openzeppelin-contracts, 2019. [Online; accessed 12-May-2020].

[12] Palkeo. Pakala. https://github.com/palkeo/pakala, 2018. [Online; accessed 3-May-2020].

[13] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss. Security analysis methods on ethereum smart contract vulnerabilities — a survey. 2020. "[Online; accessed 5-April-2020]".

[14] Remix. Remix ethereum ide. https://remix.ethereum.org, 2020. [Online; accessed 6-April-2020].

[15] S. C. Security. Smart contract weakness classification and test cases. https://swcregistry.io/, 2019. [Online; accessed 6-April-2020].

[16] P. Tsankov, A. Dan, D. Drachsley-Cohen, A. Gervais, F. Bunzli, and M. Vechev. Securify: Practical security analysis of smart contracts. 2018. "[Online; accessed 12-May-2020]".