Security Vulnerabilities in Smart Contracts as Specifications in Linear Temporal Logic

by

Indrani Ray

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2021

© Indrani Ray 2021

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Ethereum is a distributed computer with a native cryptocurrency. Like other monetary transaction based systems, a problem this platform faces is accounts and transactions being susceptible to theft and other hacks. Smart contracts (programs which run on this blockchain) can store money and initiate financial transactions. They need to be carefully studied to safeguard against threats. This is especially true before deployment, as they become immutable after. Software analysis and verification techniques are applied to study security vulnerabilities in smart contracts. Currently, there are over 35 tools that do so. Many of them directly study contracts written in high level languages such as Solidity. In this work, we similarly study contracts, but at the lower bytecode level. We focus on different classes of smart contract vulnerabilities– access control, bad randomness, denial of service, front running, integer overflow/underflow, re-entrancy, short address, time manipulation, and unchecked low-level calls. We create specifications based on linear temporal logic to describe vulnerabilities in each of these categories, and we test them against real-world contracts.

Acknowledgements

I thank Professor Mahesh Tripunitara for his supervision; professors Arie Gurfinkel and Derek Rayside for their help in thesis revision; and my family and friends for their supportive disposition(s).

Dedication

This is dedicated to my past self.

Table of Contents

1 Introduction **1**

2 Background **3**

2.1 Introduction to Ethereum . . . 3

2.1.1 Bitcoin . . . 3

2.1.2 Ethereum State Machine . . . 4

2.2 Existing Tools . . . 7

2.2.1 Oyente . . . 7

2.2.2 Zeus . . . 8

2.2.3 Maian . . . 8

2.2.4 Mythril . . . 8

2.2.5 Vandal . . . 8

2.2.6 MadMax . . . 9

2.2.7 teEther . . . 9

2.2.8 Securify . . . 9

2.2.9 Manticore . . . 10

2.2.10 eThor . . . 10

2.2.11 ETHBMC . . . 12

2.3 SmartBugs . . . 13

2.4 Model Checking . . . 15

2.4.1 Kripke Structure 15
2.4.2 Linear Temporal Logic (LTL) 15
2.4.3 nuXmv 16

3 Vulnerabilities in Smart Contracts **18**
3.1 Access Control 18
3.1.1 Description 18
3.1.2 Example 19
3.2 Bad Randomness 20
3.2.1 Description 20
3.2.2 Example 20
3.3 Denial of Service 22
3.3.1 Description 22
3.3.2 Example 22
3.4 Front Running 24
3.4.1 Description 24
3.4.2 Example 25
3.5 Integer Overflow/Underflow 26
3.5.1 Description 26
3.5.2 Example 27
3.6 Re-entrancy 28
3.6.1 Description 28
3.6.2 Example 28
3.7 Short Address 30
3.7.1 Description 30
3.7.2 Example 30
3.8 Time Manipulation 31
3.8.1 Description 31

3.8.2 Example . . . 32
3.9 Unchecked Low-Level Calls . . . 33
3.9.1 Description . . . 33
3.9.2 Example . . . 33

4 Encoding Vulnerabilities as LTL Formulae **35**
4.1 Access Control . . . 41
4.2 Bad Randomness . . . 45
4.3 Denial of Service . . . 46
4.4 Front Running . . . 48
4.5 Integer Overflow/Underflow . . . 49
4.6 Re-entrancy . . . 50
4.7 Short Address . . . 51
4.8 Time Manipulation . . . 52
4.9 Unchecked Low-Level Calls . . . 52

5 Validation **56**
5.1 Basic Approach . . . 57
5.1.1 Disclaimers . . . 57
5.1.2 Hypotheses . . . 58
5.1.3 Results . . . 59
5.2 Future Validation . . . 64

6 Conclusion **66**
6.1 Pattern Refinement . . . 66
6.2 Further Validation . . . 67
6.3 Future Inspiration . . . 67

References **69**

Chapter 1

Introduction

Ethereum is a blockchain technology containing a native cryptocurrency, Ether, and the ability to run decentralized applications (DApps) through the usage of smart contracts with frontend user interfaces [20]. The term Ethereum Virtual Machine (EVM) refers to environment containing all smart contracts and accounts. It has one canonical state that is maintained by computers running Ethereum clients, and it operates as a stack machine. We present more details about EVM in Chapter 2. The benefits of using the Ethereum platform for transactions are that users are not censored, data is public and immutable, and contracts have verifiable behavior. Nevertheless, there are drawbacks. Some drawbacks and bugs were discovered through hacks– several of which lead to monetary losses.

One of the most famous hacks, the DAO hack, happened in 2016 which resulted in 3.6 million Ether ($50 million USD) being stolen. Here is a brief summary of the hack as given by Siegel [48]. A Decentralized Autonomous Organization (DAO) was created by the company Slock.it to build "smart locks" allowing people to share cars, boats, apartments, etc. A DAO is run by smart contracts. It has an initial funding period (the time for this incident was 28 days) in which people participate in a crowdsale; after which, it begins to operate. People propose ways to spend the money, and the members who bought-in vote on the proposals. This organization had raised over $150m during its funding period. Before the members could determine projects to fund, an attacker began draining funds into another DAO with the same structure as the original.

The attackers were able to perform a sequence of calls to continuously withdraw from the DAO. The withdrawal process, as expected, sends funds to the withdrawer. However, it

triggers a special function (of the withdrawer) when doing so. This function can be edited to make further withdrawals– before the original process (which includes steps such as updating the balance after the withdrawal) is complete. The attackers made their child DAO (the withdrawal recipient) continuously withdraw from the DAO. This process is an example of a smart contract re-entrancy attack.

The DAO hack (and other real-world incidences) inspires researchers to study security vulnerabilities in smart contracts. As smart contracts can be thought of as programs, researchers implement program analysis and verification techniques to study them. In this work, we take a close look at different types of security vulnerabilities existing in Ethereum smart contracts written in the high-level language Solidity. We extract patterns at the bytecode level (expressed as human-readable opcodes) to describe these vulnerabilities, and formulate them syntactically.

This thesis is organized as follows. Chapter 2 begins with an introduction of Ethereum. The introduction is comprised of a brief discussion of Bitcoin followed by a description of the Ethereum Virtual Machine (EVM) and smart contracts. This section is followed by summaries of existing tools used in contract analysis. We then present a work, SmartBugs [31], which has guided and provided data for our work. We end the chapter with background information about model-checking. Chapter 3 presents descriptions and examples of nine types of vulnerabilities in smart contracts written in Solidity. They are access control, bad randomness, denial of service (DoS), front running, integer overflow/underflow, re-entrancy, short address, time manipulation, and unchecked low-level calls. Chapter 4 presents the crux of our work. We translate the vulnerabilities from the previous chapter into expressions written in Linear Temporal Logic (LTL). We discuss validation techniques for these in Chapter 5. We conclude with a summary of this work and plans for the next steps.

The hypothesis we are trying to show is can we meaningfully express security vulnerabilities in Ethereum smart contracts using LTL formulae based of off Ethereum Virtual Machine (EVM) opcodes [1]. The eventual goal is to show whether a smart contract is susceptible to a given vulnerability.

[1]We use LTL, instead of other temporal specification languages, for ease-of-compatibility with our model checker. This is discussed in Chapter 2.

Chapter 2

Background

2.1 Introduction to Ethereum

Ethereum is a decentralized, open-source blockchain technology. It is similar to other digital currency platforms such as Bitcoin but has a language to write smart contracts. In the following subsections, we will discuss Bitcoin, the structure of Ethereum, and other key terminology related to Ethereum.

2.1.1 Bitcoin

Bitcoin [1] allows for decentralized, digital currency on a blockchain platform. A blockchain is a set of records (blocks). Each block contains a hash of the previous block, a timestamp, and transaction data. A block is immutable– changing it would cascade changes in subsequent blocks. The ledger of Bitcoin is a state transition system– the states represent the ownership status of all existing bitcoins. A state transition takes a state and a transaction and outputs the resulting state. A transaction can be, for instance, a transfer of bitcoins. Bitcoin establishes a proof-of-work system. This is a way for the blocks to verify the appending of a new block and mining of currency.

Like Bitcoin, Ethereum is reliant on blockchain technology and has a native cryptocurrency, Ether. Instead of a distributed ledger, Ethereum is a distributed state machine.

2.1.2 Ethereum State Machine

Ethereum can be viewed as a transaction-based state machine [20]. The states are made up of accounts. There are two types– externally owned accounts and contract accounts. An externally owned account has a public-private key pair to sign and authenticate transactions; a contract account interacts with other accounts as specified in its code. Each account is identifiable by a 160-bit address and contains: a nonce (for an externally owned account, this is the number of transactions sent from its address, and for a contract account, this is the number of contract-creations made by this account), a balance (the number of Wei owned by the address of this account), a storage root (the 256-bit hash of the root node of a Merkle Patricia tree that encodes the storage contents of the account), and the code hash (the hash of the immutable EVM code that gets executed when the address of this account is called).

Transactions are actions initiated by externally-owned accounts. Transactions can either result in message calls or contract creation [51]. Moreover, the term "transaction" sometimes explicitly refers to the signed data package [20]. A transaction includes a nonce (number of transactions sent by the sender), gas price (number of Wei for all computation costs due to the execution of the transaction), gas limit (the maximum amount of gas that should be used to execute the transaction), either a 160-bit address of the message call's recipient or an empty field in the case of contract creation, an endowment (for contract creation) or amount of Wei to be transferred to the message call's recipient, a signature to determine the sender of the transaction, and an unlimited size byte array (For contract creation, this contains code corresponding to the account initialization procedure. For message call, this specifies the input data of the call.).

Smart Contracts

A smart contract is a computer program which serves as an agreement between two parties and is implemented by the consensus of the blockchain without the need for a third-party trusted authority [23]. Smart contracts are a type of account in Ethereum. They are written in high-level, object-oriented languages such as Solidity and Vyper [33, 16]. The smart contracts discussed in this work are written in the former language. Smart contracts are compiled into bytecode which is natively executed on the Ethereum virtual machine (EVM). The bytecode can be disassembled to human readable opcodes; we use the terms "bytecode", "opcode", and "instruction" interchangeably.

We examine contracts written in Solidity. Solidity is statically typed. It supports inheritance, libraries, and user-defined types and structures. Each Solidity file contains a pragma. This must be declared explicitly as Solidity syntax and compiler versions frequently evolve– consequently, some vulnerabilities become obsolete. In fact, the compiler version has changed around 30 times since January 2020. Ether and time units can be expressed in Solidity contracts using keywords such as `wei` and `seconds`. There are also special variables and functions. For example, `tx.gasprice` is used to determine the gas price of the transaction. Solidity also has ABI encoding and decoding functions, error handling functions, and cryptographic functions. For more information, please refer to the documentation [33].

Solidity contracts are compiled into EVM code. As mentioned earlier, this is expressed a a sequence of opcodes. These dictate how stack items are added or removed, how gas is reduced, how the program counter changes, how memory is read from and written to, and other state changes. More information can be found in the Yellow Paper [51].

EVM is the runtime environment for smart contracts. The machine state is defined by a tuple which contains available gas (non-negative integer), program counter (natural number whose value is at most 2^{256}), memory (word-addressed byte array), active number of words in memory (word size is 256-bit), and stack (container with maximum size of 1024). It is altered by a series of instructions. The EVM is a quasi-Turing complete machine [51]. The term *quasi* is used because computation is bounded. A system with a set of instructions is said to be Turing complete if it can simulate any Turing machine.

We support the claim of EVM being a quasi-Turing complete machine. An unlimited register machine (URM) is equivalent to a Turing machine (TM). We show that an EVM can 'model' a URM, and thereby a TM.

A URM has an infinite number of registers containing natural numbers as well as a program which is a finite list of instructions. The register contents can be altered using the four types of instructions: zero $Z(n)$, successor $S(n)$, transfer $T(m, n)$, and jump $J(m, n, q)$. In EVM, computation is done using the stack. However, the stack size is not unlimited– it can hold 1024 256-bit values. Below, we show a reduction from EVM opcodes to URM instructions. We first define the instruction then simulate it using a sequence of opcodes.

1. Z(n): Change the content of register n to 0. Proceed to next instruction.
   Assume that the nth entry in the stack is N. We can change this to 0 using the EVM

opcodes of `PUSH1 0x00 SWAP(n+1) POP`. The first opcode appends a 1-byte item of value 0 to the first position of the stack. The second opcode swaps the first and $(n+1)^{st}$ item, so the first item of the stack is N and the $(n+1)^{st}$ is 0. The final opcode removes the first item off the stack. So we have our original register, but with 0 as the nth entry.

2. S(n): Add 1 to the value of the content in register n. Proceed to next instruction.
   Let the nth entry of the stack be N and the first entry be X. We can create a successor function as follows: `SWAP(n-1) PUSH 0x01 ADD SWAP(n-1)`. The first opcode swaps X and N. Then 1 is appended to the first position of the stack. `ADD` adds the first two items of the stack. It removes them and pushes their sum to the first position of the stack. So we have $1+N$ at the first position, X in the nth, and the other items as they were initially. We finally swap $1+N$ and X. We have our original register, but with the nth item incremented by 1.

3. T(m, n): Replace the content of register n by that of register m. Proceed to next instruction.
   Assume that our stack is such that the value in position m is M, in n is N, and 1 is X. We can create a transfer function as follows: `SWAP(n-1) POP DUP(m-1) SWAP(n-1)`. This first switches X with N. The next instruction removes N from the first position of the stack. The third instruction duplicates M and pushes this duplicate onto the stack. At this point our stack has M in position 1, M in position m, and X in position n. The final instruction switches items in position 1 and n. We are left with our original stack, but with a copy of M in position n and no N.

4. J(m, n, q): If the contents of registers m and n are equal, then proceed to instruction q. Otherwise proceed to next instruction.
   Assume that our stack is such that the mth item is M and the nth item is N. Further, assume Q is the hexadecimal version of the number $q-3$. To perform this jump operation, we will do the following: `PUSH1 Q DUP(m+1) DUP(n+2) EQ ISZERO JUMPI JUMPDEST POP POP`. We begin by adding Q to the top of the stack. We then duplicate M and N. At this point, our stack has N in position 1, M in position 2, Q in position 3, and the original stack contents shifted down three positions (so the original M is in position $m+3$ and N in $n+3$). The next opcode `EQ` compares the items in the first two positions of the stack. It removes these and pushes either 1 or 0 onto the stack– 1 if they are equal and 0 otherwise. `ISZERO` changes the first item of the stack to 1 if it is 0 and to 0 otherwise. The stack is now 1 or 0 followed by Q followed by the original stack contents. We then switch the first two stack entries. The next opcode `JUMPI` conditionally alters the program counter. If the second item

of the stack is 0, then the program counter is incremented by 1. If the second item is 1, then the program counter is set to the first item of the stack (in our case, Q). We use `JUMPDEST` as a place holder for the $q - 3^{rd}$ instruction. The next two opcodes remove the first two items of the stack (either Q and 0 or Q and 1). We are left with our original stack and our program counter set to q.

We have used EVM opcodes to simulate zero, successor, transfer, and jump instructions. All computable functions can be expressed as a combination of these four instructions and computations can be done using an unlimited register of natural numbers. However, as the size of the EVM stack is finite, we cannot fully model a URM. Therefore, not all computable functions can be expressed as a combination of EVM opcodes with computations done on the EVM stack. We cannot claim that EVM is Turing complete, but we can support the statement of EVM being quasi-Turing complete [1].

2.2 Existing Tools

There are over 35 tools that analyze Ethereum smart contracts. We briefly discuss some of the prominent ones below.

2.2.1 Oyente

Oyente [39, 52] is a symbolic execution tool used to find security bugs. It takes as input the bytecode of a contract to be analyzed and the Ethereum global state (this provides the current values of contract variables). It outputs 'problematic' symbolic paths. The tool has four main parts. The first part constructs a control-flow graph of the contract. The next part explores the paths of control-flow. It produces a set of traces. Each trace is associated with a path constraint, and a Z3 solver is used to eliminate provably infeasible traces. The third part contains the bulk of the analysis. It determines whether a contract has transaction-ordering dependence, timestamp dependence, mishandled exceptions, and/or re-entrancy vulnerabilities. The last component validates whether a flaw (as mentioned in the prior segment) can actually occur from a given trace.

[1] The Ethereum Yellow Paper [51] also points out that computation is bounded by gas which also contributes to the *quasi* qualification.

2.2.2 Zeus

Zeus [37] is a symbolic model checker for Solidity-based smart contracts. The authors classify contracts into two categories: incorrect and unfair. They further split these depending on the types of vulnerabilities present in the contracts. The categories for the former are re-entrancy, unchecked `send`, failed `send`, integer overflow/underflow, and transaction state dependence. The categories for the latter are absence of logic, incorrect logic, and logically correct but unfair. We base our understanding of the vulnerabilities, in this thesis, off of their definitions in the incorrect category. The tool creates policies based on these vulnerabilities and inserts corresponding assert statements into the smart contract code. It then translates this into LLVM bitcode and uses the SeaHorn verifier to determine assertion violations. The tool is also compatible with other LLVM bitcode based verifiers such as SMACK or DIVINE.

2.2.3 Maian

Maian [44, 43] is a symbolic analyzer for smart contract execution traces. It takes as input a contract in bytecode form and a concrete starting block value. The tool then symbolically executes contract traces to find whether certain properties can be violated. It then uses a concrete validation routine to confirm concrete exploits. The goal of the tool is to be able to automatically detect three types of buggy smart contracts: suicidal (can be killed by anyone), prodigal (can send Ether to anyone), and greedy (cannot extract Ether from).

2.2.4 Mythril

Mythril [41, 45] is an analysis tool that detects security vulnerabilities in smart contracts. It uses a symbolic interpreter for EVM bytecode known as LASER. This represents smart contract execution as a space of states and path formulae (expressed in propositional logic). Mythril then uses an SMT solver to determine whether states are reachable following certain conditions. This helps determine vulnerabilities. Mythril then uses concolic testing to determine whether these vulnerabilities are exploitable.

2.2.5 Vandal

Vandal [24, 15] is a security analysis framework for Ethereum smart contracts. The tool is comprised of four parts: scraper (retrieves bytecode from the blockchain), disassembler

(translates bytecode into opcode), decompiler (translates bytecode to register transfer language to show control-flow), and extractor (translates register transfer language into logic semantic relations). The logic relations are defined in Soufflé which is based on Datalog.

2.2.6 MadMax

MadMax [35, 10] is a static program analysis framework which detects gas-based vulnerabilities in compiled smart contracts. Many vulnerabilities arise from out-of-gas exceptions. This happens primarily because a user pays for a transaction upfront. The transaction computation may require more gas than what is paid, so the transaction may get aborted. The tool utilizes the Vandal decompiler to translate the EVM bytecode to an intermediate representation. Then, it determines gas-based vulnerabilities (for instance caused by unbounded mass operations, wallet griefing, and integer overflows) and correlates them with analysis done at the bytecode level (on, for instance, program constructs, program flow, data structures, memory layout, and other characteristics).

2.2.7 teEther

teEther [18, 14] is an automatic exploitation framework. The tool first dissassembles the EVM bytecode to construct a control-flow graph. This is scanned to find EVM instructions that can be used to extract Ether from smart contracts. Critical instructions are `CALL`, `SELFDESTRUCT`, `CALLCODE`, and `DELEGATECALL`. These instructions combined with certain conditions are used to define a contract being in a vulnerable state. The tool has a path generation module which explores the paths (in the CFG) that lead to such states. A set of path constraints is created (through symbolic execution). These are then used to produce sequences of transactions attackers can perform to exploit contracts.

2.2.8 Securify

Securify [50, 12] is a security analyzer for Ethereum smart contracts. It takes as input the EVM bytecode of a contract and a set of compliance and violation security patterns written in a domain-specific language. (Security patterns can be re-entrancy bugs, locked Ether, missing input validation, unrestricted Ether flow, and others). The bytecode is initially transformed into a stackless representation in static-single assignment form. Then the tool performs decompilation. Afterwards, it infers behaviors of the contract such as data and control-flow dependencies. These facts inferred in this process are known as semantic facts.

The tool checks the security patterns against the semantic facts. It then points out the instructions that caused a violation or compliance.

2.2.9 Manticore

"Manticore is a symbolic execution framework for analyzing binaries and smart contracts" [40, 11]. The main elements of Manticore are the core engine, the native and Ethereum execution modules, SMT module, event system, and API. The core engine manages states at each point of execution (a point is after the invocation of one instruction). It implements state exploration. The Ethereum module symbolically executes smart contracts with symbolic transactions (both value and data are symbolic). These are repeatedly executed to explore the state space of a contract. The features of Manticore are: program exploration, input generation, error discovery, instrumentation, and a programmatic interface.

2.2.10 eThor

This is a sound and automated static analyzer for EVM bytecode [47, 6, 7]. The tool supports reachability properties and contract-specific functional properties. It designs a framework for the specification and implementation of static analyses based on semantic abstractions specified by a set of constrained Horn clauses (with predicate applications representing contract execution) . Its abstractions are based off of the blockchain environment, gas modelling, the memory model, and the callstack. However, note the following:

- *Blockchain environment*: As the analysis is based on the invocation of a contract in an arbitrary environment, the ever-changing execution environment and other parts of the global state are not modeled.
- *Gas modelling* this is also not analyzed. Gas does not have much of an impact on execution until it runs out.
- *Memory model*: memory is modeled as a word array.
- *Callstack* is a two-level abstraction– one level represents the original execution of the invoked contract and the other re-entrancies.
- *Execution states* are abstracted as predicate applications which include: $MState_{pc}((size, s), m, stor, cl)$ which describes a regular execution of the contract at program counter pc with stack of size $size$ and elements as described by the mapping

s (from stack positions), local memory m, the global storage of the contract at this point $stor$, and the call level cl; $Exc(cl)$ denotes that an execution of a contract halted exceptionally on call level cl; $Halt(stor, cl)$ represents a regular execution halt on call level cl with global storage being $stor$.

EVM opcodes that are abstracted include `ADD`, `CALL`, `DELEGATECALL`, `CALLCODE`, `CREATE`, and `SHA3`; however, `DELEGATECALL`, `CALLCODE` are later omitted from their analysis. The analysis supports reachability properties which capture security properties. The reachability properties are:

- Single-entrancy: Call reachability is an over-approximation of single-entrancy, because an internal transaction is initiated by the execution of a call instruction.
- Static assertion checking: Developers can add assertions (which are enforced by the compiler at runtime by throwing exceptions when they are violated using the `INVALID` opcode). This work statically checks the reachability of these opcodes.
- Semi-automated verification of contract-specific properties: $\{P\}C\{Q\}$ is a Hoare triple where P is the precondition on the execution state, C is the contract code, and Q is the postcondition that should be satisfied.

The security properties of smart contracts taken into consideration in this work are:

- If an execution state executes an annotated contract and the contract is present in the global state of the state, then the state is *strongly consistent* with the contract.
- Initial configurations that result in valid external transactions are *reachable*.
- Sometimes a contract calling another contract can be called back before completing the original transaction. This is known as re-entrancy. A contract is *single-entrant* if it cannot perform any more calls after re-entering.

The authors of this work create a framework, *HoRSt*, which takes a specification defining the static analysis and produces an encoding suitable for Z3. They use this to implement their static analyzer *eThor*.

2.2.11 ETHBMC

After surveying some static analyzers mentioned above, Frank et al. [34, 4], found that they were lacking in four categories, and proposed a symbolic execution bounded model checker. Below, we summarize each category and present the solutions from this work.

1. Inter-contract Reasoning: During contract execution, the Ethereum system allows a contract to execute (and hence interact with) others. When contracts are created by other contracts, this relationship is referred to as intra-contract. ETHBMC reasons about this as follows. The researchers model the execution of the first contract as an abstract state machine. If there is a message call to another contract during execution of this one, they create another abstract state machine for latter contract. From these, they construct execution trees which are comprised of the states of the machines. The execution tree of the initial contract forks (at the state in which the call to the second contract is made) into the final states of the second contract. This procedure can be done recursively and with multiple calls, and each possible outcome is simulated.

2. Memory: Program code gets executed on the virtual machine which provides a `memcpy` like opcode. ⌜In C, this function takes in as parameters: pointer to the destination array where the content is to be copied, pointer to the source of the data to be copied, and the number of bytes to be copied. It returns a pointer to the destination [2] ⌟. In Solidity, `CALLDATACOPY` takes some number of bytes of the call data at a specific position and copies it into a specific position in the memory [49]. To recall, a contract receives a fresh instance of memory for each message call. According to the authors of this work, this is not modeled correctly in previous works. They model memory using a memory modification graph. Each node has a unique index and a label (INIT, WRITE, COPY, SET) which corresponds to how the memory is altered upon contract execution. The resulting graph has a tree-like structure, and can be encoded into logical formulae. Note: a contract may have multiple such memory graphs based on execution memory, calldata, and storage; in such a case, the graph will have a forest-like structure. Each account's memory graph is initialized by a storage memory node upon the account's execution. The graph(s) is altered based on the instructions such as `SSTORE`, `MSTORE`, and `CALLDATALOAD`.

3. Symbolically modeling every possible outcome of this hash function is basically impossible. Solidity uses Keccak256 Function, for example, when implementing mapping datatype. This work proposes an encoding of pairs of 3-tuples. Each tuple

(KECCAK.ADR, KECCAK.LEN, KECCAK.M) contains the starting memory address, the length of the memory range, and the index of the execution memory at the time of computation respectively. The encoding takes as input in two distinct tuples, the set of path constraints, and the memory graph. It outputs a modified set of path constraints.

4. Validation: The authors of this work claim that other analysis tools over-approximate and find several false positives. Those works do not validate their findings. They address this issue by empirically validating their findings.

The tool has a symbolic executor, a detection module, and a validation module. The executor performs a breadth-first search and uses an SMT solver (Yices2, Boolector, or Z3) to assert satisfiability of a code path. They encode various types of attacks using additional path constraints in their detection module. For example, they identify contracts which can be destroyed by external attackers by flagging states with `SELFDESTRUCT` instructions. In the validation module, they generate valid transactions (using an SMT solver) for states which have feasible attack paths.

2.3 SmartBugs

We adopt two components of the work "Empirical Review of Automated Analysis Tools of 47, 587 Ethereum Smart Contracts" [31]. This paper compares nine analysis tools in detecting vulnerabilities in smart contracts. The comparison is done in two parts. The authors have amassed a dataset of 69 [2] smart contracts– referred to as $sb_{curated}$ [29]. These contracts are manually flagged as having security vulnerabilities in one of the categories from the Decentralized Application Security Project [3]. We will discuss these categories in further detail in Chapter 3. This dataset is used to analyze the precision of the analysis tools. The second dataset, sb_{wild} [30], is a set of 47518 smart contracts whose source code is available on Etherscan [5]. The researchers conclude that the tools can only find 42% of the vulnerabilities. They under-perform in the areas of access control, denial of service (DoS), and front running, and are unable to detect bad randomness and short address vulnerabilities. The tools flag 93% of actual smart contracts as having vulnerabilities (there is a high number of false positives).

[2] We use a larger set of contracts provided by the authors of this work– containing 135 contracts.

There are some limitations of this work which we include here. The first is with regards to their classification of types of vulnerabilities. The work follows DASP10 [3]. The categories here are broad. For example, one category is access control. In our study, we found this category incorporates the vulnerabilities of unprotected `SELFDESTRUCT` instruction, unprotected Ether withdrawal, and authorization through `tx.origin` [3]. DASP10 is not comprehensive. One of the DASP10 categories is unknown unknowns. This category is not well-defined. The authors include contracts they come across as potential new categories of vulnerabilities. For example, the authors of one of their surveyed tools come up with a vulnerability type known as locked Ether. The authors of this work include that in the unknown category but do not define it. We, therefore, do not study the contracts in this category.

Next, many of the tools compared do not target all of the 10 categories. Moreover, some tools compared are not designed to detect vulnerabilities. For example, as mentioned in Section 2.2.3, Maian is designed to detect suicidal, prodigal, and greedy smart contracts. It is not designed to detect, for instance, front running vulnerabilities. When testing this tool, the authors of this work accordingly found that it was unable to detect any of the DASP10 in contracts with those known vulnerabilities. Thus, not all tools are suited for the tasks attempted.

Furthermore, the choice of tools to include in the study is restricted. This is understandable as comparing a wide variety of tools is difficult. The selection bias omits some well-known tools from the study. For example, the tools examined must take contracts written in Solidity as input. This criteria excludes tools such as Vandal which take EVM bytecode. Some tools are excluded because they are not compatible with the framework created in this work.

Finally, the categorization of contracts can be questioned. The assumption that each contract only contains one type of vulnerability can be questioned. We argue, in the subsequent chapters, that a contract can be susceptible to multiple vulnerabilities of different types.

[3] This nomenclature comes from the SWC Registry [13].

2.4 Model Checking

Although not a part of this thesis, we give a brief background to model checking to help understand our ongoing work. Model checking is a way to determine whether a finite-state model of a system (such as a push-down system or timed automaton) meets a specification. Given a Kripke structure K and a temporal logic formula ϕ, a model checker decides whether $K \models \phi$ [28]. Model checking is restricted to finite state systems to mitigate the state explosion problem (the size of the system state grows exponentially as the number of state variables increases).

2.4.1 Kripke Structure

A Kripke Structure [38, 27] contains a finite, directed graph along with an interpretation and is used to represent a system. The vertices of the graph represent states and the edges are state transitions. States encapsulate information about the system at specific moments of time; transitions show gradual changes. Formally, a Kripke structure over a set of atomic propositions AP is a tuple $M = (S, I, R, L)$ where

- S: a finite set of states
- $I \subseteq S$: a set of initial states
- $R \subseteq S \times S$: a transition relation where $\forall s \in S \ \exists \ s' \in S \ \ s.t. \ (s, s') \in R$
- $L : S \to 2^{AP}$: a labelling function

2.4.2 Linear Temporal Logic (LTL)

We adopt our understanding of Linear Temporal Logic (LTL) from *Principles of Cyber-Physical Systems* [22]. LTL is a fragment of first-order logic which incorporates temporal operators. We use the following notation. Let q be a valuation over a set of typed variables, V, and let f be a Boolean expression over V. Then, q satisfies f if $q(f) = 1$: $q(f)$ represents evaluating f using the values of the variables given by q. We often need to consider an infinite sequence of valuations, a.k.a. a trace, when reasoning about a temporal logic formula. A trace ρ satisfies the Boolean expression f if the first valuation satisfies it. Temporal logic formulae can be combined using the standard $\wedge, \vee, \neg, \to$ operators. There are some additional operators as well: always $\Box$, eventually $\Diamond$, next $\bigcirc$, and until $\mathbf{U}$. The

following summarizes the satisfaction rules [22]. We are given a trace (which recall is an infinite sequence of valuations) $\rho = q_1 q_2 q_3 ...$, LTL-formulae ϕ, ϕ_1, and ϕ_2, positions j and k, and a Boolean expression f (note: a Boolean expression is also an LTL-formula).

- $(\rho, j) \models f$ if the valuation q_j satisfies the Boolean expression f.
- $(\rho, j) \models \neg\phi$ if it is not the case that $(\rho, j) \models \phi$.
- $(\rho, j) \models \phi_1 \wedge \phi_2$ if both $(\rho, j) \models \phi_1$ and $(\rho, j) \models \phi_2$.
- $(\rho, j) \models \phi_1 \vee \phi_2$ if either $(\rho, j) \models \phi_1$ or $(\rho, j) \models \phi_2$.
- $(\rho, j) \models \phi_1 \rightarrow \phi_2$ if either $(\rho, j) \models \neg\phi_1$ or $(\rho, j) \models \phi_2$.
- $(\rho, j) \models \bigcirc\phi$ if $(\rho, j+1) \models \phi$.
- $(\rho, j) \models \Box\phi$ if for every position $k \geq j, (\rho, k) \models \phi$.
- $(\rho, j) \models \Diamond\phi$ if for some position $k \geq j, (p, k) \models \phi$.
- $(\rho, j) \models \phi_1 \mathbf{U} \phi_2$ if for some position $k \geq j, (\rho, k) \models \phi_2$ and for all positions $j \leq i < k$, $(\rho, i) \models \phi_1$.

The language $\mathcal{L}$ of a Kripke structure is the set of computation paths. We say $K \models \phi$ if and only if $\forall \ell \in \mathcal{L}$, $\ell \models \phi$.

2.4.3 nuXmv

This thesis is part of an ongoing work. The validation using model checking is briefly discussed in this thesis. We adopt NuXmv for the purpose of our validation.

NuXmv is a symbolic model checker based off of NuSMV. NuSMV [26] is a model checker which supports SAT-based and BDD-based techniques. It can be used to check whether an LTL or CTL specification holds for a defined model. NuSMV recognizes six types. The Boolean type comprises *True* and *False* values. The integer type is any whole number in the range $[-2^{31} + 1, 2^{31} - 1]$. The two enumeration types *symbolic enum* and *integers-and-symbolic enum* contain symbolic constants and both integer numbers and symbolic constants respectively. The word types *unsigned* and *signed* are used to model vectors of bits thereby allowing arithmetic and bit-wise logical operations. Array types can be of

Booleans, integers, enumerations, or words. Finally, set types are expressions representing set of values– they are *Boolean set*, *integer set*, *symbolic set*, and *integers-and-symbolic set*.

NuXmv has several features. In addition to new model checking algorithms for finite-state systems, it includes symbolic algorithms for infinite-state systems (for example bounded model checking and counter-example guided abstraction refinement). It also contains an extended language for synchronous systems. This includes the new types of *integer*, *reals*, and *uninterpreted functions*. For more information, please refer to "The nuXmv Model Checker" [25].

Chapter 3

Vulnerabilities in Smart Contracts

A comprehensive set of vulnerabilities existing in smart contracts written in Solidity, with examples, can be found in the SWC Registry [13]. The vulnerabilities mentioned here correspond to common hardware and software weaknesses. However, this thesis is based on a more general categorization as given in the curated dataset of smart contracts [31] which is based off of the DASP10 [3]. However, we limit this to nine vulnerabilities, as DASP10 contains one miscellaneous category.

3.1 Access Control

3.1.1 Description

There are different types of access control issues that can exist in smart contracts. One predominant one is with respect to the special variable `tx.origin`. `tx.origin` is a global variable in Solidity which yields the address of a transaction's sender [33]. If there is a chain of contract transactions $A \rightarrow B \rightarrow C$, calling the variable `tx.origin` in C will return the address of A. Using `tx.origin` for authorization can result in an access control vulnerability (as shown in the example below). It is advised to use `msg.sender` instead of `tx.origin` for authorization [33] as the former variable yields the address of the invoker of the current call.

3.1.2 Example

There are two contracts above. `UserWallet`'s constructor sets an address variable to the address of whoever is currently calling this contract. It also has a function which sends an unsigned integer amount to the address of another contract. Before doing so, it allegedly checks the authorization of the contract initiating this transaction.

`AttackWallet`, has a constructor which creates an instance of `UserWallet` and assigns the address of the invoker as the `owner`. It also has a function which calls the `sent` function of that `UserWallet` instance. Finally, there is a function which, although not relevant in

the actual attack, helps visualize it.

Walk-Through of Attack

An attack which exploits this vulnerability (in `UserWallet`) is as follows.
Deploy `UserWallet` and then deploy `AttackWallet` using the former's address. From the second contract, invoke `getIt`. During this, the execution jumps to `sent` function (written on the left in the above diagram). Now, it should be the case that the authorization (the subsequent line) fails and the function reverts. However, that is not the case. The execution continues on (to `destination.transfer(amount)`), and the `AttackWallet` is able to pour funds into the contract specified in the `sent` portion of `getIt`– in this case, the contract that deployed `AttackWallet`. To understand the attack better, call the `seeThwart` function. This returns two addresses. The first address is one used for the `getIt` call (in this example, the parent contract) and the actual address of the contract making said call. It is important to note that if `AttackWallet` is deployed by a contract different from the one deployed by `UserWallet`, then this attack does not work.

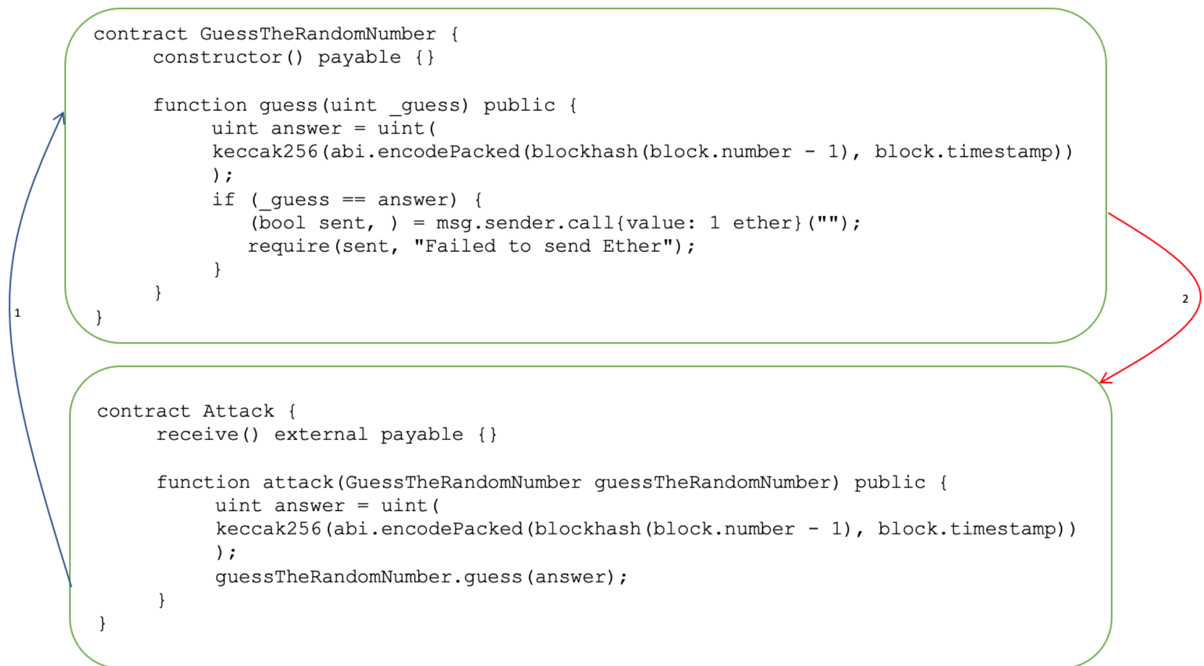
3.2 Bad Randomness

3.2.1 Description

Some contracts use and generate random numbers. An example application using such contracts is for gambling DApps which rely on the generation of pseudo-random numbers to select winners. Sometimes this random number generation is dependent on computations using hard-to-predict values such as a block's timestamp, hash, number, and/or difficulty. However, these values can be manipulated by the block's miner, and therefore the numbers generated using them are not random. Instead of relying on block characteristics as sources of randomness, using an external oracle or commitment scheme is recommended [13].

3.2.2 Example

The example below is from Tasuku Nakamura [42].



The first contract, `GuessTheRandomNumber` has a function which computes a "random" number using the block number and timestamp. If a contract calling this one can guess the "random" number, then it sends 1 ether to it. The second contract has a function that computes a number and uses it as a guess to send to the first contract.

Walk-Through of Attack

The idea behind the attack is that the second contract makes an educated guess thereby showing that the number computed is not actually random. A contract which repeatedly invokes `guess(...)` to guess the "random" number will deem it impossible to find out. However, the contract `Attack` makes an educated guess and is rewarded. Let us say `GuessTheRandomNumber` is deployed with 1 ether. An attacker can create the `Attack` contract, deploy it, and call `attack()`. Since the attacker's guess is computed the same way as the number to be guessed, it will be correct. Therefore, `GuessTheRandomNumber` will send `Attack` 1 ether.

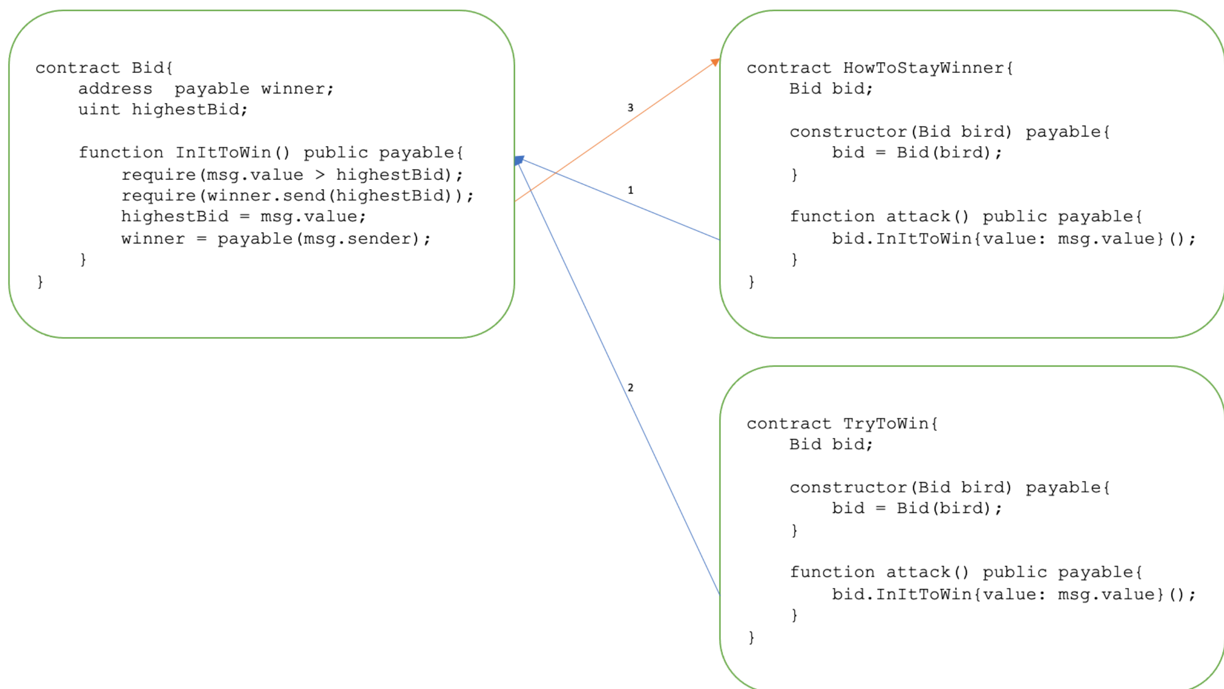
3.3 Denial of Service

3.3.1 Description

There are different types of Denial of Service (DoS) attacks. One type is called DoS with Unexpected Revert. It is also known as DoS with Failed Call [17]. We will discuss this below, but the idea behind this DoS is that external calls can fail. It is recommended to isolate external calls into individual transactions that can be initiated by the recipient of the call and to assume external calls can fail. The other type of DoS is Block Gas Limit [17]. The amount of computation done by each block is dependent on the amount of gas it has. If the gas spent by the block exceeds the amount of gas available, there can be a failed transaction and, therefore, a DoS. This DoS can be on a contract (via unbounded operations) or on the network (via block stuffing). The main remediation advice is to avoid actions that require looping over data structures [13].

3.3.2 Example

The type of DoS attack given in this example is caused by an unforeseen revert function. When a contract calls another, the second contract may have such a function which interrupts the execution of the first.



The above example has two contracts. `Bid` has address and unsigned integer variables along with a function. This function, when called by other contracts, does the following. It makes sure the amount of Wei sent with the call is more than the integer `highestBid`. If so, and if another contract had called this function previously, it allegedly refunds the previous contract. Then, it sets the `highestBid` value to be the value sent in the current message call and the `winner` to be the contract that called this function.

The second contract, `HowToStayWinner` creates an instance of a `Bid`. It has a function which calls `InItToWin` with some input Wei value.

The third contract, `TryToWin` is similar to the second. It is not needed in the attack, but helps illustrate the attack. It will be used in the discussion below.

Walk-Through of Attack

An example of a DoS attack is as follows. Deploy the first contract and the second using the first's address. From the second contract, call the `attack` function with some amount of Wei (e.g. 5 Wei). Deploy `Try to Win` similarly to the deployment of `InItToWin`. From this

third contract, invoke `attack` with some amount of Wei greater than the other contract's invocation (e.g. 6 Wei).

It should be the case that `TrytoWin` wins the bid: `InItToWin` will be refunded its bid value (5 Wei), `highestBid` will be set as the new value (6 Wei), and `winner` will be `TrytoWin`. However, these three consequences do not happen. The refund process fails, so `InItToWin` stays as the winner.

3.4 Front Running

3.4.1 Description

This category of vulnerability is sometimes referred to as Transaction Order Dependence (TOD), as Time-of-Check vs. Time-of-Use (TOCTOU), and as Race Conditions [3]. When a transaction is initiated, there is no guarantee that the transaction will run from the same state that the contract is in upon initialization of the transaction. So, if a contract has multiple transactions there is no way for the transaction senders to determine the order in which the transactions will be run (thus the contract state is not determinable). The transaction execution order is determined by block mining. Miners have the ability to influence the ordering of transactions.

According to Eskandari et al. [32], front running vulnerabilities are categorized into three types: displacement, insertion, and suppression. A displacement attack is one where a user/contract ends up making a function call with no meaningful effect, because that call was made by another (usurper). For example, if there is an auction, a malicious user can make a bid by copying the bid of another. The original bidder may end up losing the bid. An insertion attack is one in which the adversary makes a call which changes the state of a contract, so a benign call ends up running on the modified state: The original (benign) function call runs after the adversary's transaction. For instance, let there be a contract selling an asset at a given price or, ideally, higher. Assume someone places an offer with a higher price. Now assume someone else places an offer with the given price, buys the asset, and decides to sell it in the same way, but at the higher price. If the first person's transaction runs after the second's, then the first person will end up with the asset (by inadvertently purchasing from the second), and the second person would make a profit. In a suppression attack, the attacker tries to delay the other contract's function call. An example of this, not discussed here, is conducted by the first winner of Fomo3D [8].

The main fix to this problem, other than removing the importance of transaction ordering, is to protect the confidentially of the transaction by limiting the visibility [13]. This can be done through commitment schemes.

3.4.2 Example

The following example is written by Tasuku Nakamura [42].

```
contract FindThisHash{
    bytes32 public constant hash =  0x564ccaf7594d66b1eaaea24fe01f0585bf52ee70852af4eac0cc4b04711cd0e2;

    constructor() payable{}

    function solve(string memory solution) public {
        require(hash == keccak256(abi.encodePacked(solution)), “Incorrect answer”);
        (bool sent, ) = msg.sender.call{value: 10 ether}(“”);
        require(sent, “Failed to send Ether”);
    }

}
```

We are given a contract `FindThisHash` containing a 32 byte sequence value `hash`, a constructor function, and a function `solve`. What this function does is given a string, calculates the Keccak256 hash of it, and compares it to a the fixed `hash` variable. If they are equal, it sends 10 Ether to the contract invoking this function.

Walk-Through of Attack

Once this contract is deployed, it can be exploited as follows. Assume another contract, B, finds the correct string (in the case above, the string is “Ethereum”). To be rewarded, B calls `solve("Ethereum")` with some amount of gas (e.g. 10 Wei). If another contract is watching the transaction pool, it can see B's answer and also call `solve("Ethereum")` but with a higher gas price (e.g. 100 Wei). The latter contract's transaction may be mined first. So the reward goes to the second contract instead of the first.

3.5 Integer Overflow/Underflow

3.5.1 Description

In Solidity, integer types [1] are represented as at most 256-bit binary values. The largest number that can be expressed is 2^{256} and the smallest is 0 (negative numbers are represented according to the two's complement method). Arithmetic done with these integers can result in overflows and underflows. The following list summarizes the types of errors that occur [36].

- Overflow in addition, multiplication, and exponentiation.
- Overflow in $++$, $+=$, and $*=$.
- Underflow in subtraction.
- Underflow in $--$ and $-=$.
- Division or modulo by zero causes an error.
- Overflow in the expression $type(int).min/(-1)$.
- Overflow in signed to unsigned conversion or vice versa (this is because the number may not be expressible in the target type).
- Overflow in size-decreasing implicit conversion.
- Overflow in shift operations ($<<$ or $>>$).
- Compiler error on an out-of-range constant expression interpreted into a type, e.g. $(uint\, x = -1)$.

It is important to note that since Solidity 0.8.0, all arithmetic operations revert on overflow and underflow unless in unchecked mode [33].

[1] A variable of type `uint8` is a value in the range $[0, 256)$.

3.5.2 Example

As underflows and overflows can occur as results of arithmetic, an example is adding two `uint8` integers whose sum is greater than 256. This is basically the idea behind the example below.

```
contract Flow{
    mapping (address => uint8) public balance;

    function populateentry(uint8 amount) public{
        balance[msg.sender] += amount;
    }

    function transfer(address to, uint8 amount) public{
        require(balance[msg.sender] >= amount);
        balance[msg.sender] -= amount;
        balance[to] += amount;
    }
}
```

We have a contract with a mapping and two functions. The mapping is between addresses and integers. The first function allows an invoker to populate the mapping entry corresponding to his address. The second allows for a value transfer, in the mapping entries, from the invoker to another address.

Walk-Through of Attack

The exploit is as follows. We deploy this contract. Using some address A we populate a mapping entry, with say a value 255, using the `populateentry` function. Using another address B we invoke `populateentry` another value, say 4. From A we can call the `transfer` function and shift 255 to B. The value corresponding to B was 4, but it gets incremented by 255 which causes an overflow as the mapping is restricted to containing `uint8` values (which, recall, are between 0 and 255).

3.6 Re-entrancy

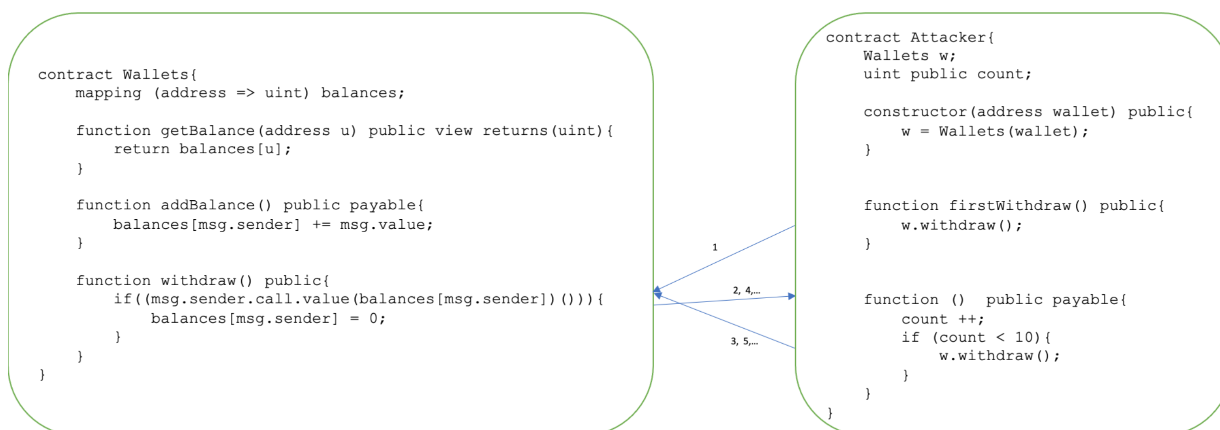
3.6.1 Description

When a contract calls another contract, the current execution is halted until the call is finished– that is, until the recipient's execution is finished. Now what may happen is that the recipient calls the former contract during its execution. In that case, the first contract is re-entered before its original execution is done. To limit the chances of re-entrancy, it is advised to follow the coding practice of using a checks-effects-interactions pattern or to use a mutex.

Most functions perform *checks* to determine who called the function, whether the call has enough Ether, whether the parameters of the function are properly-formatted, etc. The function code should be arranged to do such checks first. Next, *effects* to the state variables should be made. *Interactions* with other contracts should be done last.

The next practice is to use a mutex. “A lock or mutex is a synchronization primitive: a mechanism that enforces limits on access to a resource when there are many threads of execution” [9].

3.6.2 Example



Above are two contracts– `Wallets` and `Attacker`. The former contract contains a mapping of addresses to unsigned integers, a function which retrieves the integer corresponding to a given address in the mapping, a function which adds the amount of Wei sent in the message to the value in the mapping corresponding to the address of the message sender, and a function which allegedly withdraws the mapping value corresponding to the message sender's address. The last function is key to the re-entrancy vulnerability. It is called by `Attacker`.

The second contract `Attacker` creates an instance of `Wallets`, `w`. It also has an unsigned integer state variable: This is not important for the actual attack but helps in visualization of the attack and in limiting the number of re-entrancies (to only 10). The contract has a withdraw function and a fallback function.

Walk-Through of Attack

After both contracts are deployed, the attack begins with the invocation of `firstWithdraw` in the `Attacker` contract. The execution then jumps to the `withdraw` function of `Wallets`. For the withdrawal to be successful, the if-condition (`if((msg.sender.call...))`) must be passed successfully. The contract will receive its funds from the mapping and this amount will be cleared (in the subsequent line). However, this does not happen.

To understand what happens, parse `msg.sender.call.value(balances[msg.sender])()`. The first instruction, `msg.sender`, returns the address of the sender of the message for the current call. This keyword is followed by `call` which calls the fallback function of the invoker (in this case `msg.sender`) and forwards all available gas. Note: the `call` instruction returns a success condition; it does not return data. The next part, `value`, is the integer number of Wei sent with the message call. In this example, this amount is `balances[msg.sender]` which is the current balance of the withdrawer. Finally, note the empty set of parentheses. This signifies there not being a payload (data in the transaction).

In the context of this attack, `Attacker`'s fallback function is invoked and `Attacker` receives all available gas "on file" (albeit in this example that number was not initialized and is therefore 0). However, when `Attacker`'s fallback function is executed, there is another call back into the `withdraw` function. This happens before the initial if-condition can be passed– before the balance can be cleared. The process repeats creating a series of re-entrancies in the `withdraw` function.

3.7 Short Address

3.7.1 Description

This vulnerability is a consequence of the EVM accepting incorrectly padded arguments. Attackers can craft truncated addresses which clients may encode incorrectly in transactions. With latest compilers, this vulnerability has become obsolete. Additionally, it has not been exploited in the wild as mentioned in the $sb_{curated}$ dataset [29].

3.7.2 Example

The following example originally comes from Eric Rafaloff's analysis of the Short Address Attack, [46], but file has been taken down by the uploader.

```
contract MyToken{
    mapping (address => uint) balances;

    event Transfer(address indexed _from, address indexed _to, uint256 _value);

    function MyToken(){
        balances[tx.origin] = 10000;
    }

    function sendCoin(address to, uint amount) returns(bool sufficient){
        if (balances[msg.sender] < amount) return false;
        balances[msg.sender] -= amount;
        balances[to] += amount;
        Transfer(msg.sender, to, amount);
        return true;
    }

    function getBalance(address addr) constant returns(uint)
        return balances[addr];
    }
}
```

The contract has a mapping of addresses to integers. It also has an event with two addresses and integer parameters. Recall, "Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log" [33]. It has a

function `MyToken` which, when called, populates the mapping item corresponding to the transaction originator with 10000 gas. It also has a function which sends an integer amount to an address from the caller of the function.

Walk-Through of Attack

The exploit is as follows. Assume we have an exchange API which interacts with the `transfer` function above. However, it prepends an (expected 20-byte long) address with 12 zero-bytes to make the address input 32-bytes long. Assume a contract has an address which is 20-bytes in length with the last byte being zero, and gives the API the truncated 19-byte address along with some integer x (these values correspond to the input of the `transfer` function). When the API pads this 19-byte address, the argument becomes 31 bytes. When the API interacts with the `transfer`, the EVM will note that the input data is not properly formatted. Instead of amending the `address _to` argument, it will add a byte at the end of the `uint _amount` argument. Consequently, the transfer will be $256x$ instead of x.

3.8 Time Manipulation

3.8.1 Description

This vulnerability is also known as Block Timestamp Alteration. According to the Ethereum Yellow Paper [51], a block's timestamp is, "A scalar value equal to the reasonable output of Unix's time() at [its] inception." In Solidity, this variable is designated by `block.timestamp` or its alias `now` (although, `now` is any time in the range of the current timestamp and 900 seconds in the future). The timestamp is used to generate pseudo-random numbers[2], to trigger time-dependent events, and to control monetary transactions. However, this value can be manipulated by a block's miner. It should be noted that a block's timestamp cannot be earlier than that of its parent block (and thus cannot be altered as so), nor can it be too ahead in the future (although this is not explicitly specified in the Yellow Paper). A general rule of advice, to developers using this variable, is: "If the scale of your time-dependent event can vary by 15 seconds and maintain integrity, it is safe to use a `block.timestamp`" [17]. The following contains an example of a smart contract which uses the special variable `block.timestamp`.

[2]Generating pseudo-random numbers falls under another vulnerability category known as bad randomness. See Section 3.2.

3.8.2 Example

```
contract UsingTime{
    uint oldTime;
    mapping (uint => address) players;
    uint playernum = 0;

    function participate() public payable{
        require(block.timestamp != oldTime);
        oldTime = block.timestamp;
        playernum += 1;
        players[playernum] = msg.sender;
    }

    function raffle() public payable{
        require(block.timestamp != oldTime);
        if(block.timestamp % 4 == 0){
            address payable lucky = payable(players[playernum]);
            lucky.transfer(3500);
        }
    }
}
```

The contract above has two integer variables and a mapping between number and addresses. This contract simulates a raffle. By calling its function `participate`, a contract can join the raffle (the contract is added to the mapping). The `raffle` is designed so that a contract whose timestamp is divisible by 4 wins and is transferred some monetary value.

Walk-Through of Attack

The attack, here, is simple. Assume the contract is deployed. Lets say there are a sequence of contracts that call `participate` with varying amounts of Wei. They may or may not decide to enter the raffle. Now assume there is an attacker who knows to be eligible to win the raffle, the timestamp needs to be divisible by 4. The attacker can manipulate his timestamp to be divisible by 4. Then he can call `participate` and immediately call `raffle` (so that he is the latest player), and be transferred 3500 gas.

3.9 Unchecked Low-Level Calls

3.9.1 Description

In general, "Solidity uses state-reverting exceptions to handle errors." These exceptions undo changes made in the current call and its sub-calls. However, this exception-throwing does not happen for the low-level functions of `call`, `delegatecall`, `staticcall`, `send`, and `transfer`. Instead, these return the Boolean value `false` and continue execution when encountering errors. For example, if a contract invokes another using the `send` call, an out-of-gas exception might arise because the call is limited to 2300 gas– but execution of the receiver's fallback function, to which the call is directed, might be more expensive [37].

The main recommendation is to check the return value of a call; also, assume calls will fail and handle them accordingly. It is advised to never call into not-trusted contracts (especially with `delegatecall`).

3.9.2 Example

This example comes from the $sb_{curated}$ dataset [29].

```
contract SendBack {
     mapping (address => uint) userBalances;

     function withdrawBalance() {
          uint amountToWithdraw = userBalances[msg.sender];
          userBalances[msg.sender] = 0;
          msg.sender.send(amountToWithdraw);
     }
}
```

The contract above has a mapping of `address` to `uint` types. It also has a function which, upon invocation, transfers the integer value corresponding to the address of the invoker (in the mapping structure) to the contract calling this function.

Walk-Through of Attack

The following is not an attack scenario. Rather it is an example of how this vulnerability can be exploited, and consequently, how this contract can be misused. Assume we have another contract which creates an instance of `SendBack` that is not payable. Furthermore, assume this contract has a function which invokes `withdrawBalance` on the created instance of `SendBack` [3]. After deploying both contracts, call the function of the second. Upon doing so, the execution will jump to `withdrawBalance`. When the execution hits `msg.sender.send(amountToWithdraw`, what happens is that return value of this function is `false` as the caller of this function (the second contract we created) is not payable. Therefore, the changes to the contract will be updated (such as the mapping value being set to 0). The return value in that special function should have been checked, and changes to the contract should have been halted.

Disclaimer: this type of exploit can be avoided depending on the compiler version. For example, on the Remix platform, compiler version `0.8.7+commit.e28d00a7` preemptively issues a warning saying, "TypeError: 'send' and 'transfer' are only available for objects of type 'address payable', not 'address'."

[3] To understand this construction, regard other vulnerabilities (such as DoS).

Chapter 4

Encoding Vulnerabilities as LTL Formulae

In the previous chapter, we discussed nine classes of vulnerabilities in Ethereum smart contracts written in Solidity. In programs, some bugs might be present due to poor programming practices such as mishandling of arithmetic operations (division by zero). Others may exist inherently within a contract such as repeated invocation of a function or lack of a termination condition. Other vulnerabilities arise from attacks. To distinguish between these categories, some inspection needs to be done regarding what manifests a vulnerability. The tools mentioned in the background section examine smart contracts. They do so by directly analyzing Solidity code or by creating symbolic abstractions to reason about them. Instead, we base our analysis at the EVM bytecode level given the assumption below.

Claim: Given the assumption that contracts written at higher languages such as Solidity and Vyper are compiled to the assembly level to be processed by the EVM, vulnerabilities that exist in a contract can be expressed through execution steps as a series of EVM opcodes.

Our approach is as follows. For each category, we attempt to isolate causalities between the vulnerability and written Solidity code. For example, some contracts use built-in functions and special variables which may be invoked or misused in an attack. We attempt to express these according to related EVM opcodes. We then examine the contracts within the given category in $sb_{curated}$ [29]. There are 18 access control, 8 bad randomness, 6 DoS, 4 front running, 15 integer overflow/underflow, 31 re-entrancy, 1 short address, 5 time ma-

nipulation, and 53 unchecked low-level call contracts. The contracts here are annotated to show where the vulnerability lies. Using the Remix platform [21], we debug the compiled contract– isolating the instances in which the vulnerability arises. We trace the EVM opcodes corresponding to these segments. Below, is an example of this process.

Let us take the special function in Solidity `<address>.call(...)`. For reference, please see the documentation [33]. The contract, wherein this lies, will call the contract specified by `<address>`. If successful, the contract forwards all available gas to it (this can also be specified in the call), and the function returns the Boolean value `true`. Otherwise, the function returns `false`. Based on its functionality, we can surmise that EVM opcodes related to this function include `GAS` and `CALL`. We can further create an experimental contract where there is only one function that specifically calls another contract to gather more opcodes– but we leave this intermediate step out in this example. Now, let us look at an actual contract from $sb_{curated}$. This contract is supposed to demonstrate an unchecked call (flagged by the comment).

```
contract ReturnValue {

  function callchecked(address callee) public {
    require(callee.call());
  }

  function callnotchecked(address callee) public {
     // <yes> <report> UNCHECKED_LL_CALLS
    callee.call();
  }
}
```

We deploy this contract and invoke the second function with, for example, the address of the contract. The execution trace corresponding to this is `DUP1 ISZERO PUSH2 0xA0 JUMPI` followed by:

```
JUMPDEST POP PUSH2 0xD5 PUSH1 0x4 DUP1 CALLDATASIZE SUB DUP2 ADD SWAP1
DUP1 DUP1 CALLDATALOAD PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
AND SWAP1 PUSH1 0x20 ADD SWAP1 SWAP3 SWAP2 SWAP1 POP POP POP PUSH2 0x110
JUMP
```

Then the special function (flagged line) is executed:

```
JUMPDEST DUP1 PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND
PUSH1 0x40 MLOAD PUSH1 0x0 PUSH1 0x40 MLOAD DUP1 DUP4 SUB DUP2 PUSH1 0x0
DUP7 GAS CALL
```

At this point, the execution stalls. Once the call is complete, it resumes and ends with `SWAP2 POP POP POP POP JUMP` then `JUMPDEST STOP`.

On the other hand, if we deploy the above contract and invoke the first function with the address of the contract, we get the following trace: `DUP1 ISZERO PUSH2 0x5D JUMPI` followed by:

```
JUMPDEST POP PUSH2 0x92 PUSH1 0x4 DUP1 CALLDATASIZE SUB DUP2 ADD SWAP1
DUP1 DUP1 CALLDATALOAD PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
AND SWAP1 PUSH1 0x20 ADD SWAP1 SWAP3 SWAP2 SWAP1 POP POP POP PUSH2 0xD7
JUMP
```

Then:

```
JUMPDEST DUP1 PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND
PUSH1 0x40 MLOAD PUSH1 0x0 PUSH1 0x40 MLOAD DUP1 DUP4 SUB DUP2 PUSH1 0x0
DUP7 GAS CALL SWAP2 POP POP ISZERO ISZERO PUSH2 0x10D JUMPI PUSH1 0x0
DUP1 REVERT
```

In this case, the `require` statement failed. Note that it can also be passed successfully [1].

Both of these functions make the call to another contract; their executions proceed as

```
DUP1 PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF AND PUSH1 0x40
MLOAD PUSH1 0x0 PUSH1 0x40 MLOAD DUP1 DUP4 SUB DUP2 PUSH1 0x0 DUP7 GAS
CALL
```

[1] We temporarily omit our analysis of this case here, but discuss it in Section 9

However, the execution resumes differently in checked and unchecked calls. We can express the unchecked call in two ways. The above sequence of opcodes followed by the second sequence in the second function (corresponding to the return from an unchecked call). Or the above sequence followed by the lack of the second sequence in the first function (corresponding to the return from a checked call). For this example, let us consider the latter. Although we lose some information, we can simplify the second sequence to be `PUSH1 0x0 DUP1 REVERT` as this only arises from the `require` statement which occurs if the call is checked.

For the conversion to LTL we make the following assumptions. First, we take opcodes themselves as LTL formulae. We take a trace ρ to be a sequence of opcodes. We make this decision, because if we take it as all possible sequences of opcodes [2], then we would rather use branching-time temporal logic. We extend our finite-length execution trace to an infinite trace by stuttering: We will repeat the last opcode. A valuation q_j is an opcode. In this work, we are only expressing EVM instructions. However, note that we could also track and express changes to the stack, the memory, etc.[3] based on the instructions as specified in Appendix H of the Yellow Paper [51]. Our expression does not account for explicit operand values.

Our structure symbolizes how a contract changes upon its execution. The initial state is the input contract (along with corresponding storage, balance, etc.). The transitions are determined by the contract code: The contract changes after each instruction is executed.

Recall the unchecked call example above. We have two statements `DUP1 PUSH20 ... CALL` and `PUSH1 DUP1 REVERT`. We will express both of these as follows:

$$\phi_1 = DUP1 \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \bigcirc(PUSH1 \wedge \bigcirc (PUSH1 \wedge \bigcirc(MLOAD \wedge \bigcirc(DUP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(SUB \wedge \bigcirc(DUP2 \wedge \bigcirc(PUSH1 \wedge \bigcirc (DUP7 \wedge \bigcirc(GAS \wedge \bigcirc(CALL))))))))))))))$$

and

$$\phi_2 = \Diamond(PUSH1 \wedge \bigcirc(DUP1 \wedge \bigcirc(REVERT))).$$

We need to be careful in how we combine these two. One way is $\phi_1 \wedge \neg\Diamond\phi_2$. Then,

[2]For example, a require statement can pass or fail. There are two traces corresponding to it.
[3]We discuss this in Chapter 6

$(\rho, j) \models (\phi_1 \wedge \neg\phi_2)$ means that at time j, the trace satisfies both ϕ_1 and ϕ_2. The first satisfaction makes sense:

$$
\begin{aligned}
&(\rho, j) \models DUP1 \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND...(CALL)...)) \\
\iff &(\rho, j) \models DUP1 \textbf{ and } (\rho, j) \models \bigcirc(PUSH20 \wedge \bigcirc(AND...(CALL)...)) \\
\iff &(\rho, j) \models DUP1 \textbf{ and } (\rho, j+1) \models (PUSH20 \wedge \bigcirc(AND...(CALL)...)) \\
\iff &(\rho, j) \models DUP1 \textbf{ and } (\rho, j+1) \models PUSH20 \textbf{ and } (\rho, j+1) \models \bigcirc(AND...(CALL)...) \\
&\vdots \\
\iff &(\rho, j) \models DUP1 \textbf{ and } (\rho, j+1) \models PUSH20 \textbf{ and } ... \textbf{ and } (\rho, j+15) \models CALL
\end{aligned}
$$

The second satisfaction is:

$$(\rho, j) \models \neg(\Diamond(PUSH1 \wedge \bigcirc(DUP1 \wedge \bigcirc(REVERT))))$$

Which means it is not the case that

$$
\begin{aligned}
&(\rho, j) \models \Diamond(PUSH1 \wedge \bigcirc(DUP1 \wedge \bigcirc(REVERT)) \\
\iff &for \ some \ k \geq j, (\rho, k) \models (PUSH1 \wedge \bigcirc(DUP1 \wedge \bigcirc(REVERT)) \\
&\vdots \\
\iff &(\rho, k) \models PUSH1 \textbf{ and } (\rho, k+1) \models DUP1 \textbf{ and } (\rho, k+2) \models REVERT
\end{aligned}
$$

Now the above works for the given contract but we have to be careful. Suppose the function `callnotchecked` has a subsequent require statement which reverts. We would then see the first sequence as well as the revert sequence. Consequently, we might mis-classify the contract as not having an unchecked call. Considerations such as this one are taken into account when developing explicit LTL-formulae corresponding to vulnerabilities. In our future work, we may use a different type of temporal specification language to bound the operators (so the eventually in this example is after 8 steps). Additionally, we will justify the correctness of the formulae we come up with.

We do the above process for 18 access control, 8 bad randomness, 6 denial of service, 4 front running, 15 integer overflow/underflow, 15 re-entrancy, 1 short address, 5 time manipulation, and 10 unchecked low-level call contracts from the $sb_{curated}$ dataset. We extract similarities as patterns in each category. We summarize our findings in the following sections.

Before we proceed, we present a few points to keep in mind. The first is with regards to *halting condition*. In Ethereum, execution terminates normally when encountering `RETURN`, `REVERT`, `STOP`, or `SELFDESTRUCT`. Exceptional halting happens when there is insufficient gas, insufficient stack items, the instruction is invalid, the jump destination is invalid, the stack size is larger than 1024, or when attempting state modification in a static call [51]. Opcodes that may be encountered during exceptional halting include `CREATE`, `CREATE2`, `CALL`, `STATICCALL`, `DELEGATECALL`, `CALLCODE`, `SSTORE`, `JUMPI`, `JUMP`, `JUMPDEST`, and `INVALID`. Some LTL formulae below are described with a terminating condition. For example, the sequence `JUMPDEST STOP` or `PUSH1 DUP1 REVERT` (some formulae show the lack thereof of the latter). The lack of a terminating sequence in an expression does not imply that execution does not halt. In some cases, an argument can be made to not include a terminating sequence in the LTL expression– for instance, $\Diamond JUMPDEST \wedge \bigcirc(STOP)$ might be attainable in several contracts regardless of whether they contain a specific vulnerability whose characterization includes this expression.

The next point is that a contract 'containing' a vulnerability specification is not necessarily vulnerable: There are false positives. This is relevant, for example, in the integer overflow/underflow category (Section 4.5). As mentioned in Section 3.5, several of these errors are consequences of arithmetic. Let us have a contract which adds two numbers. Given that our expression captures addition, we may see that such a contract is flagged as having an integer overflow/underflow vulnerability. However, say in practice, the contract is used to add $1+1$. This addition will not cause an overflow or underflow. Whether or not we can support the claim that a vulnerable contract 'contains' the respective specification for the vulnerability will be discussed in the next chapter.

Finally, several formulae below were created by observing the $sb_{curated}$ dataset. The contracts here are annotated to show where the vulnerability occurs. Some formulae capture the annotated part. They do not capture how the vulnerability is exploited or is manifested. This is apparent, for example, in front running (Section 4.4). This vulnerability arises in contract-to-contract or off-chain situations. Expressing these interactions by observing contract code is not feasible, so we can only express the parts of the contract that are interacted with.

We have a few more remarks with regards to our process. It is important to note that the LTL formulae were created to find patterns, but not all vulnerable contracts follow these patterns. For example, by looking at all access control contracts in $sb_{curated}$ (18 unique contracts) we find that 3 of them follow a specific pattern. Our LTL formula corresponding

to this group will therefore 'fail' on the remaining 15 contracts. One might argue for complete coverage– such as 15 formulae for 15 contracts (or another surjection from formulae to contracts). But, our goal in this work is to describe vulnerabilities not individual contracts. Another remark although we have tried to keep our notation standard, some LTL formulae are aesthetically inconsistent. Some appear as a long continuous formula while others are broken into smaller ones. Furthermore, in some places, we write $formula = operator(...)$ and in others we write $formula = (...)$ and then present it as $operator formula$ – although they appear to be different, they are not in the context of this work.

4.1 Access Control

In Chapter 2, we discuss a type of access control issue stemming from the usage of the special variable, in Solidity, `tx.origin` for authorization. Most contracts with this type of vulnerability check the value of this variable against the value of `msg.sender`. This is the address of the one initiating the current call. The contracts containing this type of vulnerability have the pattern $((\phi_1 \wedge \bigcirc\phi_2) \vee (\phi_1 \wedge \bigcirc\phi_2 \wedge \bigcirc \bigcirc \phi_2)) \wedge \Diamond(\phi_3 \vee (\phi_3 \wedge \bigcirc\phi_4))$ where

$$\begin{aligned}
\phi_1 &= ORIGIN \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(EQ))) \\
\phi_2 &= ISZERO \\
\phi_3 &= PUSH2 \wedge \bigcirc(JUMPI) \\
\phi_4 &= PUSH1 \wedge \bigcirc(DUP1 \wedge \bigcirc(REVERT))
\end{aligned}$$

This characterization is quite broad. Let us walk through what that above formula means. We have a conjunction of two formulae at a given time j. Let us parse the shorter one (following $\Diamond$). This says that eventually for some $k \geq j$, our trace either satisfies ϕ_3 or ϕ_3 followed by ϕ_4. Note: we could have left our formula to end at ϕ_3– the ϕ_4 is some additional information which we will talk about. ϕ_4 corresponds to a sequence of opcodes `PUSH1 DUP1 REVERT`. This sequence manifests in the execution trace when there is some assertion that fails. For the `tx.origin` vulnerability, this arises when the contract requires some variable representing ownership to be equal to `tx.origin`. ϕ_3, by itself, does not have much of a meaningful semantic. However, ϕ_1, ϕ_2, and ϕ_3 taken together do. They correspond to part of an execution trace which checks if `tx.origin` is equal to some value. Although the formula above can perhaps be simplified to $(\phi_1 \wedge \Diamond\phi_3)$ with $\phi_1 = ORIGIN \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(EQ \wedge \bigcirc(ISZERO))))$, we keep the lengthier expression (as it is more detailed).

If our contract compares the value of `tx.origin` to the value of `msg.sender` our formula above will contain some instance of `CALLER`. If these two values are directly compared, our formula will look like `CALLER PUSH20 AND ORIGIN...`. If there is some intermediate step such as an intermediate variable being set to `msg.sender` and that variable being compared to `tx.origin`, then the execution trace will contain `CALLER` followed by some sequence of opcodes and then eventually `...PUSH20 AND ORIGIN...`. In these cases, we can modify our formula to be: $\phi_5 \wedge \Diamond(((\phi_1 \wedge \bigcirc\phi_2) \vee (\phi_1 \wedge \bigcirc\phi_2 \wedge \bigcirc\bigcirc\phi_2)) \wedge \Diamond(\phi_3 \vee (\phi_3 \wedge \bigcirc\phi_4)))$ where

$$\phi_1 = PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(ORIGIN \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(EQ)))))$$
$$\phi_5 = CALLER$$

and the other formulae are as they were above. Again, an argument can be made to condense this as done above.

Another type of access control vulnerability is "incorrect constructor name." According to the Solidity documentation [33], this optional function is declared with the `constructor` keyword and is executed upon contract creation. The author of the contracts with this vulnerability in $sb_{curated}$ claims: the constructor function needs to have the same name as the contract class; otherwise, it is just a callable function. However, according to the Solidity documentation, this was true prior to Solidity version 0.4.22. Recent compilers do not detect any issues with this. The formula for this vulnerability is:

$$\begin{aligned}\phi_5 = {} & ISZERO \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMPI \wedge \Diamond(JUMPDEST \wedge \bigcirc(POP \wedge \bigcirc(PUSH2 \wedge \\ & \bigcirc(PUSH2 \wedge \bigcirc(JUMP \wedge \Diamond(JUMPDEST \wedge \bigcirc(CALLER \wedge \bigcirc(PUSH1 \wedge \bigcirc(DUP1 \wedge \\ & \bigcirc(PUSH2 \wedge \bigcirc(EXP \wedge \bigcirc(DUP2 \wedge \bigcirc(SLOAD \wedge \bigcirc(DUP2 \wedge \bigcirc(PUSH20 \wedge \bigcirc(MUL \wedge \\ & \bigcirc(NOT \wedge \bigcirc(AND \wedge \bigcirc(SWAP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(MUL \wedge \\ & \bigcirc(OR \wedge \bigcirc(SWAP1 \wedge \bigcirc(SSTORE \wedge \bigcirc(POP \wedge \bigcirc(JUMP \wedge \Diamond(JUMPDEST \wedge \\ & \bigcirc(STOP))))))))))))))))))))))))))))))))\end{aligned}$$

The above formula captures more information than needed for this vulnerability as it encapsulates what the function does in addition to what the function is. The sequence beginning with `CALLER` describes the computation of setting a private variable to the value of `msg.sender`. Recall, `msg.sender` yields the payable address of the originator of the current call. If we want to isolate the function apart from what happens inside of it, we can say

$$\begin{aligned}\phi_5 = {} & ISZERO \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMPI \wedge \Diamond(JUMPDEST \wedge \bigcirc(POP \wedge \bigcirc(PUSH2 \wedge \\ & \bigcirc(PUSH2 \wedge \bigcirc(JUMP \wedge \Diamond(JUMPDEST \wedge \bigcirc(STOP))))))))).\end{aligned}$$

But, this may occur as evidence of any function[4] – not necessarily an incorrect constructor name one. In which case, every contract containing a function will be flagged as having this vulnerability. To reduce the number of false positives (arising from using these LTL formulae to detect vulnerabilities), it is better to use the original ϕ_5.

One more vulnerability in the access control category is known as "missing modifier." The formula corresponding to the contracts with this in $sb_{curated}$ is:

$$\begin{aligned}\phi_6 = {} & DUP3 \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(EQ \wedge \bigcirc(ISZERO \wedge \bigcirc(ISZERO \wedge \bigcirc(ISZERO \wedge \\ & \bigcirc(PUSH2 \wedge \Diamond(JUMPI \wedge \Diamond(JUMPDEST \wedge \bigcirc(CALLER \wedge \bigcirc(PUSH1 \wedge \\ & \bigcirc(PUSH1 \wedge \bigcirc(DUP5 \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \\ & \bigcirc(DUP2 \wedge \bigcirc(MSTORE \wedge \bigcirc(PUSH1 \wedge \bigcirc(ADD \wedge \bigcirc(SWAP1 \wedge \bigcirc(DUP2 \wedge \\ & \bigcirc(MSTORE \wedge \bigcirc(PUSH1 \wedge \bigcirc(ADD \wedge \bigcirc(PUSH1 \wedge \bigcirc(SHA3 \wedge \bigcirc(PUSH1 \wedge \\ & \bigcirc(PUSH2 \wedge \bigcirc(EXP \wedge \bigcirc(DUP2 \wedge \bigcirc(SLOAD \wedge \bigcirc(DUP2 \wedge \bigcirc(PUSH20 \wedge \\ & \bigcirc(MUL \wedge \bigcirc(NOT \wedge \bigcirc(AND \wedge \bigcirc(SWAP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(PUSH20 \wedge \\ & \bigcirc(AND \wedge \bigcirc(MUL \wedge \bigcirc(OR \wedge \bigcirc(SWAP1 \wedge \bigcirc(SSTORE \wedge \bigcirc(POP \wedge \bigcirc(PUSH1 \wedge \\ & \bigcirc(SWAP1 \wedge \bigcirc(POP \wedge \bigcirc(SWAP2 \wedge \bigcirc(SWAP1 \wedge \bigcirc(POP \wedge \bigcirc(JUMP \wedge \\ & \bigcirc(JUMPDEST)))\end{aligned}$$

According to the Solidity documentation, there is an issue known as modifier overriding. To prevent this keywords `virtual` and `override` must be used. Here is an example:

```
contract Base
{
    modifier foo() virtual {_;}
}

contract Inherited is Base
{
    modifier foo() override {_;}
}
```

[4]In fact, if we debug, for example, the `withdraw` function of the vulnerable contract "incorrect_constructor_name1" from $sb_{curated}$, we see this pattern– even though that function is not related to this vulnerability.

If the keywords are not used, the contracts are not able to be compiled. With that being said, the execution trace for both *modifier* lines is:

```
PUSH1 0x40 MSTORE CALLVALUE DUP1 ISZERO PUSH1 0xF JUMPI JUMPDEST POP
PUSH1 0x3F DUP1 PUSH1 0x1D PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN
```

This also appears when one of the modifiers is `modifier foo() {-;}`. In fact, the entire compiled code for both contracts appears similar. The opcodes of the three types of modifiers differ from opcode number [5] 34 onwards: The first contract is `SWAP1...BASEFEE...`, the second is `EXP...SELFBALANCE...`, and the third (not shown above) is `EXTCODESIZE SHL BASEFEE...`. We can perhaps create a formula for missing modifiers from the sequence above and the opcodes in the three sets, but that is left as future work.

Another access control issue arises when a contract has a function to kill itself using `selfdestruct(<address>)`. This kills the current contract and forwards remaining gas to the given address. Our corresponding formula can be just $\phi_7 = SELFDESTRUCT$. Note: if the address is `msg.sender`, then our formula can be modified to $\phi_7 = CALLER \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(SELFDESTRUCT)))$.

The $sb_{curated}$ dataset has some contracts flagged as vulnerable to access control issues, because they contain a private variable which can be modified by using a public function. We have yet to come up with a formula to express this as it depends on what happens in the function. The contracts in $sb_{curated}$ follow $\phi_8 \wedge \Diamond(\phi_9 \wedge \Diamond(\phi_{10} \wedge \bigcirc(\phi_{11} \wedge \Diamond(\phi_{12}))))$

$$\phi_8 = ISZERO \wedge \bigcirc((PUSH1 \vee PUSH2) \wedge \bigcirc(JUMPI))$$

$$\phi_9 = JUMPDEST \wedge \bigcirc(POP \wedge \bigcirc(((PUSH1 \wedge \bigcirc PUSH1) \vee (PUSH2 \wedge \bigcirc PUSH2)) \wedge \Diamond(JUMP)))$$

$$\phi_{10} = JUMPDEST \wedge \bigcirc((DUP1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(DUP1))) \vee (CALLER \wedge \bigcirc(PUSH1 \wedge \bigcirc(PUSH1))))$$

$$\phi_{11} = PUSH2 \wedge \bigcirc(EXP \wedge \bigcirc(DUP2 \wedge \bigcirc(SLOAD \wedge \bigcirc(DUP2 \wedge \bigcirc(PUSH20 \wedge \bigcirc(MUL \wedge \bigcirc(NOT \wedge \bigcirc(AND \wedge \bigcirc(SWAP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(MUL \wedge \bigcirc(OR \wedge \bigcirc(SWAP1 \wedge \bigcirc(SSTORE \wedge \bigcirc(POP \wedge \Diamond(JUMP))))))))))))))))))$$

[5] Enumerate every opcode in the compiled code regardless of execution path.

$$\phi_{12} = JUMPDEST \wedge \bigcirc(STOP)$$

If we were to define a contract with this vulnerability as any contract which sets a private variable using a public function, our formula would be more broad– again we leave this as future work.

4.2 Bad Randomness

The specification corresponding to this vulnerability is quite loose. The formula, ϕ_1, corresponds to setting a variable to be a random value using special variables `block.timestamp`, `block.number`, or `blockhash`.

$$\phi_1 = PUSH1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MSTORE \wedge \bigcirc((TIMESTAMP \vee NUMBER \vee BLOCKHASH) \wedge \bigcirc(PUSH1 \wedge \bigcirc(STORE \wedge \bigcirc(CALLVALUE))))))$$

Several contracts do arithmetic calculations using these special variables to compute a random number. We have yet to come up with an expression to capture this as it varies from a contract-to-contract basis. For example, the contract below, from $sb_{curated}$, uses addition, multiplication, division, and modular arithmetic.

```
contract RandomNumberGenerator {
  uint256 private salt =  block.timestamp;

  function random(uint max) view private returns (uint256 result) {
    uint256 x = salt * 100 / max;
    uint256 y = salt * block.number / (salt % 5);
    uint256 seed = block.number / 3 + (salt % 300) + y;
    uint256 h = uint256(blockhash(seed));
    return uint256((h / x)) % max + 1;
  }

}
```

It has been flagged for bad randomness four times. The execution traces for these differ: for the variables `salt`, `y`, `seed`, and `h`. The execution traces of the last three are

```
PUSH1 0x5 PUSH1 0x0 SLOAD DUP2 ISZERO ISZERO PUSH2 0xB0 JUMPI JUMPDEST
```

```
MOD NUMBER PUSH1 0x0 SLOAD MUL DUP2 ISZERO ISZERO PUSH2 0xBF JUMPI
JUMPDEST DIV SWAP3 POP
```

; then

```
DUP3 PUSH2 0x12C PUSH1 0x0 SLOAD DUP2 ISZERO ISZERO PUSH2 0xD2 JUMPI
JUMPDEST MOD PUSH1 0x3 NUMBER DUP2 ISZERO ISZERO PUSH2 0xDF JUMPI
JUMPDEST DIV ADD ADD SWAP2 POP
```

; and

```
DUP2 BLOCKHASH PUSH1 0x1 SWAP1 DIV SWAP1 POP
```

A generic formula for this vulnerability could be $\phi_2 = (TIMESTAMP \vee NUMBER \vee BLOCKHASH)$, but that would flag any contract using one of these three variables– the contract might not even be using them to generate a random number.

4.3 Denial of Service

As mentioned in Chapter 2, there are different types of attacks. Our first formula corresponds to a re-entrancy type DoS.

$$\begin{aligned}
\phi_1 = PUSH1 \wedge \bigcirc(DUP1 \wedge \bigcirc(SWAP1 \wedge \bigcirc(SLOAD \wedge \bigcirc(SWAP1 \wedge \bigcirc(PUSH2 \wedge \\
\bigcirc(EXP \wedge \bigcirc(SWAP1 \wedge \bigcirc(DIV \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(PUSH20 \wedge \\
\bigcirc(AND \wedge \bigcirc(PUSH2 \wedge \bigcirc(PUSH1 \wedge \bigcirc(SLOAD \wedge \bigcirc(DUP2 \wedge \bigcirc(SLOAD \wedge \\
\bigcirc(SWAP1 \wedge \bigcirc(DUP2 \wedge \bigcirc(ISZERO \wedge \bigcirc(MUL \wedge \bigcirc(SWAP1 \wedge \bigcirc(PUSH1 \wedge \\
\bigcirc(MLOAD \wedge \bigcirc(PUSH1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \bigcirc(DUP1 \wedge \bigcirc(DUP4 \wedge \\
\bigcirc(SUB \wedge \bigcirc(DUP2 \wedge \bigcirc(DUP6 \wedge \bigcirc(DUP9 \wedge \bigcirc(DUP9 \wedge \bigcirc(CALL \wedge \\
\Diamond(SWAP4 \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(ISZERO \wedge \\
\bigcirc(ISZERO \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMPI \wedge \bigcirc(PUSH1 \wedge \bigcirc(DUP1 \wedge \\
\bigcirc(REVERT)))
\end{aligned}$$

This vulnerability arises from the failure of a refund process– gas needs to be forwarded to a contract, but is being blocked/prevented from being sent. This obstacle manifests

through the last three opcodes of the formula above. The first eleven opcodes `PUSH1 ... AND` take an address to which gas will be sent. Opcodes 14 through 19, `PUSH2 ... SWAP1` process the amount of gas to be sent. The rest of the formula until `CALL` deals with sending this amount to that address. After that transfer is made, the execution resumes with `SWAP4` until the failure mentioned above.

The second type of DoS arises from entering a special type of if-statement. The formula is $\phi_2 \wedge \neg \bigcirc \phi_3$ or $\phi_2 \wedge \bigcirc \phi_4$ where

$$\begin{aligned}\phi_2 = {} & PUSH2 \wedge \bigcirc(PUSH1 \wedge \bigcirc(DUP1 \wedge \bigcirc(SLOAD \wedge \bigcirc(SWAP1 \wedge \\ & \bigcirc(POP \wedge \bigcirc((GT \vee LT) \wedge \bigcirc(ISZERO \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMPI)))))))))\end{aligned}$$

$$\phi_3 = JUMPDEST \wedge \bigcirc(JUMP \wedge \bigcirc(JUMPDEST \wedge \bigcirc(STOP)))$$

$$\phi_4 = PUSH1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \bigcirc(SWAP1 ...)))$$

Our characterization is redundant as ϕ_4 is an instance of $\neg\phi_3$; however, it may be the case the second formula detects fewer false positives than the first.

The third type of DoS comes from an exhaustion of gas. The formula is $\phi_6 \mathbf{U}(\phi_7 \vee \phi_8)$ or $\phi_6 \wedge \bigcirc(\phi_8 \wedge \neg\Diamond\phi_9)$ where:

$$\begin{aligned}\phi_6 = {} & DUP1 \wedge \bigcirc(DUP1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(ADD \wedge \bigcirc(SWAP2 \wedge \\ & \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(PUSH2 \wedge \Diamond(JUMP \wedge \Diamond(JUMPDEST \wedge \\ & \bigcirc(DUP2 \wedge \bigcirc(DUP2 \wedge \bigcirc(LT \wedge \bigcirc(ISZERO \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMPI \wedge \\ & \bigcirc(PUSH1 \wedge \bigcirc(DUP1 \wedge \bigcirc(SLOAD \wedge \bigcirc(SWAP1 \wedge \bigcirc(POP \wedge \bigcirc(PUSH1 \wedge \\ & \bigcirc(SLOAD \wedge \bigcirc(EQ \wedge \bigcirc(ISZERO \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMPI \wedge \bigcirc(PUSH1 \wedge \\ & \bigcirc(DUP1 \wedge \bigcirc(DUP2 \wedge \bigcirc(DUP2 \wedge \bigcirc(DUP1 \wedge \bigcirc(SLOAD \wedge \bigcirc(SWAP1 \wedge \\ & \bigcirc(POP \wedge \bigcirc(ADD \wedge \bigcirc(SWAP2 \wedge \bigcirc(POP \wedge \bigcirc(DUP2 \wedge \bigcirc(PUSH2 \wedge \\ & \bigcirc(SWAP2 \wedge \bigcirc(SWAP1 \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMP \wedge \Diamond(JUMPDEST \wedge \\ & \bigcirc(POP \wedge \bigcirc(JUMPDEST \wedge \bigcirc(DUP3 \wedge \bigcirc(PUSH1 \wedge \bigcirc(PUSH1 \wedge \\ & \bigcirc(DUP1 \wedge \bigcirc(DUP2 \wedge \bigcirc(SLOAD \wedge \bigcirc(DUP1 \wedge \bigcirc(SWAP3 \wedge \bigcirc(SWAP2 \wedge \\ & \bigcirc(SWAP1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(ADD \wedge \bigcirc(SWAP2 \wedge \bigcirc(SWAP1 \wedge \bigcirc(POP \wedge \\ & \bigcirc(SSTORE \wedge \bigcirc(DUP2 \wedge \bigcirc(SLOAD \wedge \bigcirc(DUP2 \wedge \bigcirc(LT \wedge \bigcirc(ISZERO \wedge \\ & \bigcirc(ISZERO \wedge \bigcirc(PUSH2 \wedge \\ & \bigcirc(JUMPI))\end{aligned}$$

$$\phi_7 = INVALID$$

$$\phi_8 = JUMPDEST \wedge \bigcirc(SWAP1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MSTORE \wedge \bigcirc(PUSH1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(SHA3 \wedge \bigcirc(ADD \wedge \bigcirc(DUP2 \wedge \bigcirc(SWAP1 \wedge \bigcirc(SSTORE \wedge \bigcirc(POP))))))))))$$

$$\phi_9 = POP \wedge \bigcirc(POP \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMP \wedge \bigcirc(JUMPDEST \wedge \bigcirc(DUP2 \wedge \bigcirc(DUP2 \wedge \bigcirc(LT \wedge \bigcirc(ISZERO \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMPI \wedge \bigcirc(JUMPDEST \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(JUMP \wedge \bigcirc(JUMPDEST \wedge \bigcirc(STOP))))))))))))))))$$

The pattern of ϕ_6 comes from the continuous execution of a loop. When the loop is exited, the execution continues with ϕ_9. So if ϕ_9 is not present, the loop has not been exited. Alternately, if we encounter the `INVALID` instruction, ϕ_7, after ϕ_6, we have run out of gas (hence, there will be a denial of service). We use the **U** operator to show the repeated existence of ϕ_6.

4.4 Front Running

It is difficult to express this vulnerability in LTL as transaction ordering is not a characteristic of the contract, but rather how contracts interact with other members of the blockchain. The contracts in $sb_{curated}$ susceptible to this bug have either a `require` or an `if` statement that, when passed, make the contract vulnerable. These statements end with `EQ ISZERO ISZERO PUSH2 JUMPI`– sometimes, there are three `ISZERO`s. Sometimes these are preceded by, but not necessarily immediately, `CALLER` or `CALLVALUE`. Depending on the smart contracts, this string might be preceded by other computations (such as comparing the input of a function to a fixed value). There is no succinct way of describing all computations, so we will keep our formula loose (additionally, the opcodes `PUSH1` and `DUP4` show up in these computations). So, a starting point, corresponding to the require statement, is

$$\phi_1 = (CALLVALUE \vee CALLER \vee (PUSH1 \wedge \Diamond DUP4)) \wedge \Diamond(EQ \wedge \bigcirc(ISZERO \wedge \bigcirc(ISZERO \wedge \Diamond(PUSH2 \wedge \bigcirc(JUMPI \wedge \Diamond(JUMPDEST)))))))$$

Note: instead of $\Diamond(JUMPDEST)$, we could have also said $\neg \bigcirc (PUSH1 \wedge \bigcirc(DUP1 \wedge \bigcirc(REVERT)))$. Then, the contracts have some type of transfer function. As we see in other vulnerabilities, transfer functions can be expressed as a sequence of opcodes. In our

case, we have

$$\phi_2 = SWAP1 \wedge \bigcirc(DUP2 \wedge \bigcirc(ISZERO \wedge \bigcirc(MUL \wedge \bigcirc(SWAP1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \bigcirc(PUSH1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \bigcirc(DUP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(SUB \wedge \bigcirc(DUP2 \wedge \bigcirc(DUP6 \wedge \bigcirc(DUP9 \wedge \bigcirc(DUP9 \wedge \bigcirc(CALL)))))))))))))))))$$

All together, we have $\phi_1 \wedge \Diamond\phi_2$

Another front running expression is:

$$\phi_3 = ISZERO \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMPI \wedge \Diamond(JUMPDEST \wedge \bigcirc(POP \wedge \bigcirc(PUSH2 \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMP \wedge \Diamond(JUMPDEST \wedge \bigcirc(CALLER \wedge \bigcirc(PUSH1 \wedge \bigcirc(DUP1 \wedge \bigcirc(PUSH2 \wedge \bigcirc(EXP \wedge \bigcirc(DUP2 \wedge \bigcirc(SLOAD \wedge \bigcirc(DUP2 \wedge \bigcirc(PUSH20 \wedge \bigcirc(MUL \wedge \bigcirc(NOT \wedge \bigcirc(AND \wedge \bigcirc(SWAP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(MUL \wedge \bigcirc(OR \wedge \bigcirc(SWAP1 \wedge \bigcirc(SSTORE \wedge \bigcirc(POP \wedge \bigcirc(JUMP \wedge \Diamond(JUMPDEST \wedge \bigcirc(STOP))))))))))))))))))))))))))))))))$$

The first three opcodes, as we have seen, are not explicitly related to this vulnerability but arise when functions are entered in smart contracts. The subsequent opcodes arise when a private structure is changed. In Solidity, new types can be defined through the usage of a `Struct` datatype. A contract in $sb_{curated}$ that follows this pattern has a structure which is comprised of tuples of addresses and integers. It has a function which sets the address to `msg.sender` and the integer to the given input of the function.

4.5 Integer Overflow/Underflow

We have, at the minimum, $\phi_1 \wedge \Diamond(\phi_2 \wedge \Diamond\phi_3)$ where:

$$\phi_1 = PUSH1$$

$$\phi_2 = DUP3 \wedge \bigcirc(DUP3 \wedge \bigcirc(SLOAD \wedge \bigcirc((ADD \vee SUB \vee MUL \vee DIV) \wedge \bigcirc(SWAP3 \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(DUP2 \wedge \bigcirc(SWAP1 \wedge \bigcirc(SSTORE \wedge \bigcirc(POP))))))))))$$

$$\phi_3 = JUMPDEST \wedge \bigcirc(STOP)$$

However, some contracts also have an optional `DUP1` followed by one or two `PUSH1` then optionally `SHA3`, `DUP1`, or `DUP2`, then sometimes `PUSH1` and `DUP1`, followed by the above formula. This is too complicated to express, and is already captured by the above formula, so we do not write another LTL formula for it. In fact, we can perhaps further tighten the expression to just $\phi_2 \wedge \Diamond\phi_3$ as `PUSH1` is a popular instruction, so the chance of it preceding the rest of the expression anywhere (e.g. by being the very first instruction for a contract) is highly likely. The same argument can be made about $JUMPDEST \wedge \bigcirc(STOP)$s succeeding which we have kept for the sake of consistency with other vulnerabilities.

The contracts in $sb_{curated}$ with this vulnerability have arithmetic operations. In most cases, some value– such as a variable or mapping entry– is being changed. There is no evidence of other types of overflow and underflow vulnerabilities in $sb_{curated}$. We could try to extrapolate some more patterns as follows:

Consider the shift operations. Create a dummy contract such as:

```
contract Shift{
    uint y;

    function overflow(uint256 x) public{
        y = x << 255;
    }

}
```

If we deploy this and call `overflow(2)`, we see the trace `DUP2 SWAP1 SHL PUSH1 0x0 DUP2 SWAP1 SSTORE POP POP JUMP`. If we create another dummy contract like the above but with right shift, our trace is `DUP2 SWAP1 SHR PUSH1 0x0 DUP2 SWAP1 SSTORE POP POP JUMP`. Our LTL formula will reflect this with one part being $SHL \vee SHR$. We can make similar extrapolations for the other categories.

4.6 Re-entrancy

Re-entrancy is $(\phi_1 \vee (\phi_2 \wedge \Diamond\phi_1)) \wedge \bigcirc\neg\phi_3$ where

$$\phi_1 = PUSH1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \bigcirc(DUP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(SUB \wedge \bigcirc(DUP2 \wedge \bigcirc(DUP6 \wedge \bigcirc(DUP8 \wedge \bigcirc(GAS \wedge \bigcirc(CALL))))))))))$$

$$\phi_2 = (CALLER \vee CALLVALUE)$$

$$\phi_3 = SWAP3 \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(POP)))$$

As articulated in other sections, we can perhaps eliminate the need for ϕ_2 as the main commonality is ϕ_1. The opcodes in ϕ_2 occur in specific instances of re-entrancy– for example, `CALLER` occurs when the current contract is invoking another specified by `msg.sender`. The opcodes of `GAS` and `CALL` are especially important in this vulnerability. They appear when a contract makes a call to another and forwards some amount of gas. Most contracts with re-entrancy contain a line like this: `<address>.call.value(<uint>)`. In this line, the contract calls the fallback function of the contract specified by the given address. Note: a contract's fallback function is executed whenever the contract receives Ether– it is also the 'default' function that is executed if the caller contract calls a function that is not available. The contract sends an integer number of Wei [6] with the message. Eventually, the `call` instruction returns a success condition. Re-entrancy happens when the contract is entered again from the called contract (before the first execution finishes). The original execution should eventually resume with ϕ_3. If it does not, then perhaps the contract has been re-entered. We have used the $\bigcirc$ operator instead of the $\Diamond$ operator to trace the original contract's execution (even though there may be intermediate execution steps belonging to the called contract). As a side note, generally ϕ_3 is followed by one or two `ISZERO` then `PUSH2 JUMPI`, but we have left it out to shorten the formula somewhat.

4.7 Short Address

Below is the formula corresponding to the flagged part of the one contract with this vulnerability in $sb_{curated}$. The expression below does not justly describe the vulnerability, as the vulnerability arises from manipulation done off-chain. Instead it describes a function which sends some integer amount to a mapping entry corresponding to an address. According to the annotations of $sb_{curated}$, if the transfer is successful, the vulnerability exists– we express it as a failure of reverting $\neg\phi_1$ where

$$\phi_1 = DUP1 \wedge \bigcirc(ISZERO \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMPI \wedge \bigcirc(PUSH1 \wedge \bigcirc(DUP1 \wedge \bigcirc(REVERT))))))$$

It should be emphasized once more that there only exists one contract with this vulnerability. Furthermore, most debuggers, such as the one built into Remix, prevent this vulnerability from taking place by warning users about input not being properly formatted.

[6] Wei is the smallest denomination of currency in Ethereum. 1 Ether = 10^{18} Wei.

4.8 Time Manipulation

Time manipulation is another broad category. The contracts in $sb_{curated}$ with this vulnerability contain the keyword `block.timestamp`. In a contract, when some variable is set to this value, our formula is

$$\phi_1 = TIMESTAMP \wedge \bigcirc(PUSH1 \wedge \bigcirc(DUP2 \wedge \bigcirc(SWAP1 \wedge \bigcirc(SSTORE \wedge \bigcirc(POP)))))$$

If there is a function where this value is returned (not set to a variable), then we have

$$\phi_2 = TIMESTAMP \wedge \bigcirc(SWAP1 \wedge \bigcirc(POP \wedge \bigcirc(SWAP1))).$$

Actually, if we have a return line, such as `return block.timestamp >= 1546300800`, the trace contains more opcodes like `TIMESTAMP LT ISZERO SWAP1 POP SWAP1`. So, we change our formula to be more flexible:

$$\phi_2 = TIMESTAMP \wedge \Diamond(SWAP1 \wedge \bigcirc(POP \wedge \bigcirc(SWAP1))).$$

Generally contracts do some computation using this value. For example, the contract entitled "time_manipulation" in $sb_{curated}$ has a line flagged for this vulnerability:
`var random = uint(sha3(block.timestamp))% 2;`
The trace corresponding to this is

```
TIMESTAMP PUSH1 0x40 MLOAD DUP1 DUP3 DUP2 MSTORE PUSH1 0x20 ADD SWAP2 POP
POP PUSH1 0x40 MLOAD DUP1 SWAP2 SUB SWAP1 KECCAK256 PUSH1 0x1 SWAP1 DIV
DUP2 ISZERO ISZERO PUSH2 0x156 JUMPI (not INVALID) JUMPDEST MOD SWAP1 POP
```

A, perhaps not meaningful, characterization of this vulnerability may just be the existence of `TIMESTAMP`.

4.9 Unchecked Low-Level Calls

Most of the $sb_{curated}$ contracts in this category have the keyword `call`. There are some contracts with the keyword `delegatecall` in the access control section; they display this vulnerability too. There are similarities between this category and re-entrancy (Section

4.6), as the latter is sometimes caused by the former.

Our first formula is:

$$\begin{aligned}\phi_1 = DUP1 \land \bigcirc(PUSH20 \land \bigcirc(AND \land \Diamond(PUSH1 \land \bigcirc(MLOAD \land \bigcirc(PUSH1 \land \\ \bigcirc(PUSH1 \land \bigcirc(MLOAD \land \bigcirc(PUSH1 \land \bigcirc(PUSH1 \land \bigcirc(MLOAD \land \bigcirc(DUP1 \land \\ \bigcirc(DUP4 \land \bigcirc(SUB \land \bigcirc(DUP2 \land \bigcirc(PUSH1 \land \bigcirc(DUP7 \land \bigcirc(GAS \land \\ \bigcirc(CALL \land \Diamond(SWAP2 \land \bigcirc(POP \land \bigcirc(POP \land \bigcirc(POP \land \bigcirc(POP \land \\ \bigcirc(JUMP \land \bigcirc(JUMPDEST \land \bigcirc(STOP))))))))))))))))))))))))))\end{aligned}$$

This arises with an `<address>.call(<bytes>)` method. This is a low-level call with a given payload of bytes of memory. It forwards all available gas to the contract called. The return is a Boolean value along with bytes of memory. This method bypasses type checking, function existence checking, and argument packing [33]. As mentioned in the example walk-through, we can also create a formula incorporating the lack of revert. This would look similar, but after $CALL$, we would include the lack of an eventual $PUSH1 \land \bigcirc(DUP1 \land \bigcirc(REVERT))$. In fact, to be more specific, we can include some more details in our revert statement.
An expression, such as: $\phi_{11} \land \Diamond(\phi_{12} \lor \neg\phi_{13})$ where

$$\begin{aligned}\phi_{11} = DUP1 \land \bigcirc(PUSH20 \land \bigcirc(AND \land \Diamond(PUSH1 \land \bigcirc(MLOAD \land \bigcirc(PUSH1 \land \\ \bigcirc(PUSH1 \land \bigcirc(MLOAD \land \bigcirc(PUSH1 \land \bigcirc(PUSH1 \land \bigcirc(MLOAD \land \bigcirc(DUP1 \land \bigcirc(DUP4 \land \\ \bigcirc(SUB \land \bigcirc(DUP2 \land \bigcirc(PUSH1 \land \bigcirc(DUP7 \land \bigcirc(GAS \land \bigcirc(CALL))))))))))))))))))\end{aligned}$$

$$\begin{aligned}\phi_{12} = SWAP2 \land \bigcirc(POP \land \bigcirc(POP \land \bigcirc(POP \land \bigcirc(POP \land \bigcirc(JUMP \land \bigcirc(JUMPDEST \land \\ \bigcirc(STOP)))))))\end{aligned}$$

$$\begin{aligned}\phi_{13} = SWAP2 \land \bigcirc(POP \land \bigcirc(POP \land \bigcirc(ISZERO \land \bigcirc(ISZERO \land \bigcirc(PUSH1 \land \bigcirc(JUMPI \land \\ \bigcirc(PUSH1 \land \bigcirc(DUP1 \land \bigcirc(REVERT)))))))))\end{aligned}$$

is redundant. Furthermore, a revert statement (like ϕ_{13} arises from an assertion failing. The case of the assertion passing needs to be included in the so-called redundant formula. In other terms, we could write $\phi_{11} \land \Diamond(\phi_{12} \lor \neg(\phi_{13} \lor \phi_{14}))$ where

$$\begin{aligned}\phi_{14} = SWAP2 \land \bigcirc(POP \land \bigcirc(POP \land \bigcirc(ISZERO \land \bigcirc(ISZERO \land \bigcirc(PUSH1 \land \bigcirc(JUMPI \land \\ \bigcirc(JUMPDEST \land \bigcirc(POP \land \bigcirc(JUMP \land \bigcirc(JUMPDEST \land \bigcirc(STOP))))))))))).\end{aligned}$$

Again, this is redundant. We, therefore, express our formulae below in the style of $\phi_{11} \wedge \Diamond\phi_{12}$.

When we have `<address>.send(<uint>)` or `<address>.transfer(<uint>)`, our formula is :

$$\phi_2 = (CALLER \vee DUP1) \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(PUSH2 \wedge \bigcirc((PUSH1 \vee DUP3)\wedge \bigcirc(SWAP1 \wedge \bigcirc(DUP2 \wedge \bigcirc(ISZERO \wedge \bigcirc(MUL \wedge \bigcirc(SWAP1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD\wedge \bigcirc(PUSH1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \bigcirc(DUP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(SUB \wedge \bigcirc(DUP2\wedge \bigcirc(DUP6 \wedge \bigcirc(DUP9 \wedge \bigcirc(DUP9 \wedge \bigcirc(CALL \wedge \Diamond(SWAP4 \wedge \bigcirc(POP \wedge \bigcirc(POP\wedge \bigcirc(POP \wedge \bigcirc(POP)))))))))))))))))))))))))))$$

Another example of this vulnerability (akin to ϕ_1 above is):

$$\phi_3 = PUSH1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \Diamond(DUP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(SUB\wedge \bigcirc(DUP2 \wedge \bigcirc((PUSH1 \vee DUP6) \wedge \bigcirc(DUP8 \wedge \bigcirc(GAS \wedge \bigcirc(CALL \wedge \bigcirc(SWAP3\wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \Diamond(JUMP \wedge \bigcirc(JUMPDEST\wedge \bigcirc(STOP))))))))))))))))))$$

This is also a manifestation of the `call` function. However, it includes some explicit bytes of data.

Some contracts with the `delegatecall` keyword (such as 'Fibonacci' in the access control set) have the following:

$$\phi_4 = PUSH1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \bigcirc(DUP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(SUB\wedge \bigcirc(DUP2 \wedge \bigcirc(DUP7 \wedge \bigcirc(GAS \wedge \bigcirc(DELEGATECALL \wedge \bigcirc(SWAP3 \wedge \bigcirc(POP\wedge \bigcirc(POP \wedge \bigcirc(POP)))))))))))))$$

The main opcode above is `DELEGATECALL`. This call variant allows a contract to dynamically load and execute code from another contract [33].

We have one more formula for a contract with `delegatecall`, but it is more complicated

than the other formulae above. $\phi_5 \wedge \bigcirc(\phi_6 \wedge \bigcirc(\phi_7 \wedge \bigcirc(\phi_6 \wedge \Diamond(\phi_8 \wedge \bigcirc\neg\phi_9))))$.

$$\begin{aligned}\phi_5 = {} & DUP2 \wedge \bigcirc(PUSH20 \wedge \bigcirc(AND \wedge \bigcirc(DUP2 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \\ & \bigcirc(DUP1 \wedge \bigcirc(DUP3 \wedge \bigcirc(DUP1 \wedge \bigcirc(MLOAD \wedge \bigcirc(SWAP1 \wedge \bigcirc(PUSH1 \wedge \\ & \bigcirc(ADD \wedge \bigcirc(SWAP1 \wedge \bigcirc(DUP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(DUP4 \wedge \bigcirc(PUSH1)))))))))))))))))\end{aligned}$$

$$\begin{aligned}\phi_6 = {} & JUMPDEST \wedge \bigcirc(DUP4 \wedge \bigcirc(DUP2 \wedge \bigcirc(LT \wedge \bigcirc(ISZERO \wedge \bigcirc(PUSH2 \wedge \\ & \bigcirc(JUMPI))))))\end{aligned}$$

$$\begin{aligned}\phi_7 = {} & DUP1 \wedge \bigcirc(DUP3 \wedge \bigcirc(ADD \wedge \bigcirc(MLOAD \wedge \bigcirc(DUP2 \wedge \bigcirc(DUP5 \wedge \\ & \bigcirc(ADD \wedge \bigcirc(MSTORE \wedge \bigcirc(PUSH1 \wedge \bigcirc(DUP2 \wedge \bigcirc(ADD \wedge \bigcirc(SWAP1 \wedge \\ & \bigcirc(POP \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMP))))))))))))))\end{aligned}$$

$$\begin{aligned}\phi_8 = {} & JUMPDEST \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(SWAP1 \wedge \\ & \bigcirc(POP \wedge \bigcirc(SWAP1 \wedge \bigcirc(DUP2 \wedge \bigcirc(ADD \wedge \bigcirc(SWAP1 \wedge \\ & \bigcirc(PUSH1 \wedge \bigcirc(AND \wedge \bigcirc(DUP1 \wedge \bigcirc(ISZERO \wedge \bigcirc(PUSH2 \wedge \bigcirc(JUMPI \wedge \\ & \bigcirc(DUP1 \wedge \bigcirc(DUP3 \wedge \bigcirc(SUB \wedge \bigcirc(DUP1 \wedge \bigcirc(MLOAD \wedge \bigcirc(PUSH1 \wedge \\ & \bigcirc(DUP4 \wedge \bigcirc(PUSH1 \wedge \bigcirc(SUB \wedge \bigcirc(PUSH2 \wedge \bigcirc(EXP \wedge \bigcirc(SUB \wedge \\ & \bigcirc(NOT \wedge \bigcirc(AND \wedge \bigcirc(DUP2 \wedge \bigcirc(MSTORE \wedge \bigcirc(PUSH1 \wedge \bigcirc(ADD \wedge \\ & \bigcirc(SWAP2 \wedge \bigcirc(POP \wedge \bigcirc(JUMPDEST \wedge \bigcirc(POP \wedge \bigcirc(SWAP2 \wedge \bigcirc(POP \wedge \\ & \bigcirc(POP \wedge \bigcirc(PUSH1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \bigcirc(DUP1 \wedge \bigcirc(DUP4 \wedge \\ & \bigcirc(SUB \wedge \bigcirc(DUP2 \wedge \bigcirc(DUP6 \wedge \bigcirc(GAS \wedge \bigcirc(DELEGATECALL \wedge \bigcirc(SWAP2 \wedge \\ & \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(ISZERO \wedge \bigcirc(ISZERO \wedge \bigcirc(PUSH2 \wedge \\ & \bigcirc(JUMPI))\end{aligned}$$

$$\phi_9 = PUSH1 \wedge \bigcirc(DUP1 \wedge \bigcirc(REVERT))$$

As this is a lengthy formula, it is most likely specific to only a few contracts. A more apt pattern would be a slice of it such as $PUSH1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \bigcirc(DUP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(SUB \wedge \bigcirc(DUP2 \wedge \bigcirc(DUP6 \wedge \bigcirc(GAS \wedge \bigcirc(DELEGATECAL)))))))))$.

We did not observe any contracts with `staticcall`. However, it can be surmised that they have similar patterns as some above (but with the keyword `STATICCALL`).

Chapter 5

Validation

In the previous chapter, there were several expressions corresponding to vulnerabilities. Although the expressions were derived directly from contracts with the given vulnerabilities, their accuracy in vulnerability detection/identification needs to be studied. In this chapter, we discuss a basic validation approach and future plans for more-thorough approach. We corroborate our findings with the $sb_{curated}$ dataset [1]. Based on our results, we will refine the expressions, and test them on the sb_{wild} dataset– but this is left as future work. The question we are trying to answer through the validation process is: *Given a smart contract and a vulnerability specification, does the contract contain that vulnerability?*
We define the following terms in the context of this study:

- False positive: A contract does not contain a specific vulnerability, but our specification detects one.
- False negative: A contract contains a specific vulnerability, but our specification does not detect one.
- True positive: A contract contains a specific vulnerability, and our specification detects one.
- True negative: A contract does not contain a specific vulnerability, and our specification does not detect one.

[1]We omit short address from our set as there is only one contract. Also, we eliminate the miscellaneous category.

5.1 Basic Approach

The first approach is quite rudimentary. We break our LTL expressions into sequences of opcodes, and we search for these sequences in the compiled code of contracts (which is represented in the human-readable EVM opcode form). We use the Remix [21] and SolC [19] platforms to compile the contracts– the versions automatically adjust depending on the pragma specified by the contract [2].

5.1.1 Disclaimers

Before we present the results of this analysis, we present a few disclaimers. As in other programming languages, the compiled code of contracts written in Solidity reflect multiple execution paths. For example, if the contract has an assertion, both successful and unsuccessful passing of the assertion are reflected in the compiled code. Let us say we have a formula $(A \wedge \bigcirc \neg B)$. If A' and B' are the corresponding opcode sequences to A and B, we want the last term of A to not be followed by the first term of B. However, the compiled code might show otherwise. Likewise, if we want $A \wedge \bigcirc C$, with C' being the opcode sequence corresponding to C, we might see $A'B'C'$ in our compiled code.

Moreover, some vulnerabilities arise from a contract's interaction with another. Some LTL formulae given above are written to show interruptions in contract execution. For example, in the re-entrancy specification, we want `PUSH1 PUSH1 MLOAD DUP1 DUP4 SUB DUP2 DUP6 DUP8 GAS CALL` to not be followed by `SWAP3 POP POP POP`. But, in the compiled code of a contract, we see `PUSH1 PUSH1 MLOAD DUP1 DUP4 SUB DUP2 DUP6 DUP8 GAS CALL SWAP3 POP POP POP`. This result seemingly contradicts our specification. We need to, therefore, loosen our specification by only searching for the first sequence. Loosening our specifications might increase our number of false positives.

Furthermore, succeeding execution steps do not manifest sequentially in the compiled code. The concept of 'eventually' is, therefore, difficult to model. Let us have the LTL formula $A\Diamond B$, and let A' and B' be the sequences of opcodes corresponding to A and B. In our approach, we will scan the contract for A' and for B'. We might find both A' and B', but

[2] The following contracts were unable to be compiled: `parity_wallet_bug_1.sol` from the access control category, `smart_billions.sol` from the bad randomness category, and `0xe09b1ab8111c2729a76f16de96bc86a7af837928.sol` and `0x19cf8481ea15427a98ba3cdd6d9e14690011ab10.sol` from the unchecked low-level calls category.

they might just be two unrelated sequences in the compiled code.

We base our analysis on the assumption that the contracts in $sb_{curated}$ only have one vulnerability each as per the classifications of the work [31].

5.1.2 Hypotheses

Below are two hypotheses we have regarding our validation.

1. The mapping from LTL formulae to vulnerable contracts is not a surjection. In our approach, we extract common patterns amongst contracts in a given category to base our respective formulae off of. Not all members of that category follow the derived patterns. The corresponding specifications, therefore, will not cover all contracts– we will have false negatives. To eliminate such false negatives, it can be argued– for example– to create an LTL formula for each contract in a category and have the disjunction of these be the overall specification. But this brute-force approach defeats the purpose of this work.

2. False positives are expected by design when checking all contracts with all specifications. For example, one unchecked low-level call specification is based off of the built-in Solidity function `<address>.delegatecall(<bytes>)`. The contract below has this function but is in the access control category.

```
contract Proxy {
    address owner;

    constructor() public {
        owner = msg.sender;
    }

    function forward(address callee, bytes _data) public {
        require(callee.delegatecall(_data));
    }
}
```

   Moreover, some specifications are quite broad– such as the integer overflow/underflow one. As mentioned in Section 4.5, many contracts in $sb_{curated}$ which display this

vulnerability have simple arithmetic. One, for example, has a variable initialized to 0 and a function which takes an integer input and adds it to that variable. The specification captures contracts which change a value using basic arithmetic. Consequently, any contract that does so is flagged as being vulnerable to integer overflow/underflow. For example, this contract from the re-entrancy category has a line which subtracts some value from an integer mapping entry.

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(balances[msg.sender] >= _weiToWithdraw);
        require(_weiToWithdraw <= withdrawalLimit);
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

5.1.3 Results

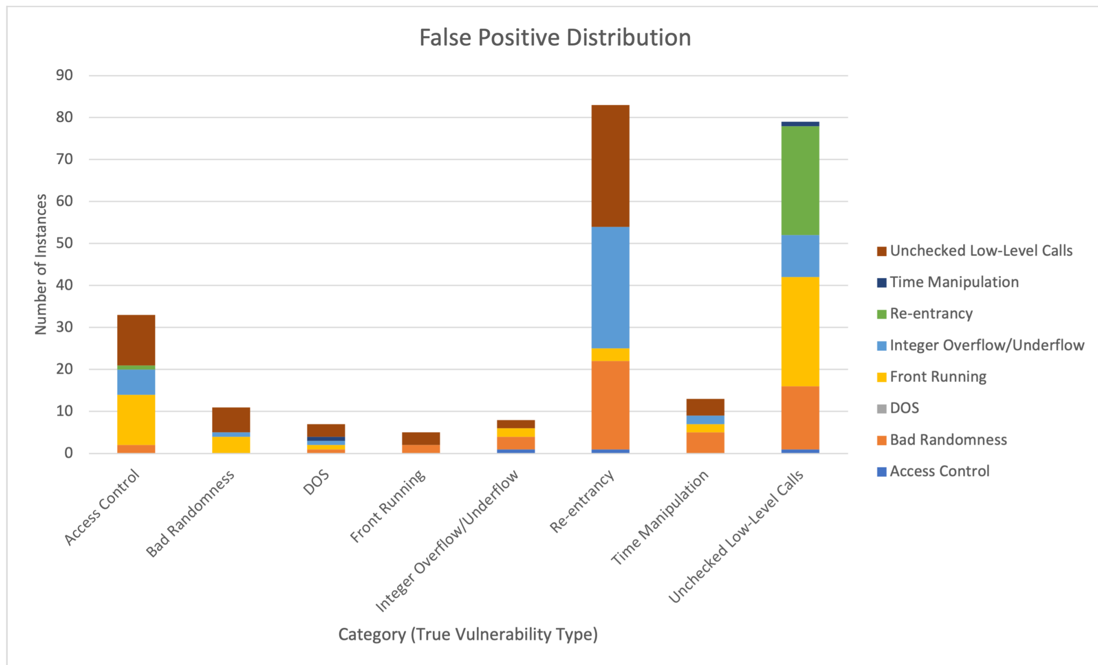
Before we present the results, we mention the following. In the modified $sb_{curated}$ dataset, the access control category has 17 contracts, bad randomness has 7 contracts, DoS has 6 contracts, front running has 4 contracts, integer overflow/underflow has 15 contracts, re-entrancy has 31 contracts, time manipulation has 5 contracts, and the unchecked low-level calls category has 50 contracts. Additionally, we have omitted some specifications– corresponding to the future work ones mentioned in Chapter 4 and to the single-opcode ones (such as `TIMESTAMP`).

We correctly identify (the true positives are) 8 access control, 7 bad randomness, 3 DoS, 4 front running, 13 integer overflow/underflow, 29 re-entrancy, 3 time manipulation, and 50 unchecked low-level calls. Thereby, the false negatives are 9 access control, 0 bad randomness, 3 DoS, 0 front running, 2 integer overflow/underflow, 2 re-entrancy, 2 time

manipulation, and 0 unchecked low-level calls.

We search for each type of vulnerability in all the contracts in the modified $sb_{curated}$ dataset mentioned above, and find 239 false positives and 706 true negatives. The graphs below show the distribution of these two sets.

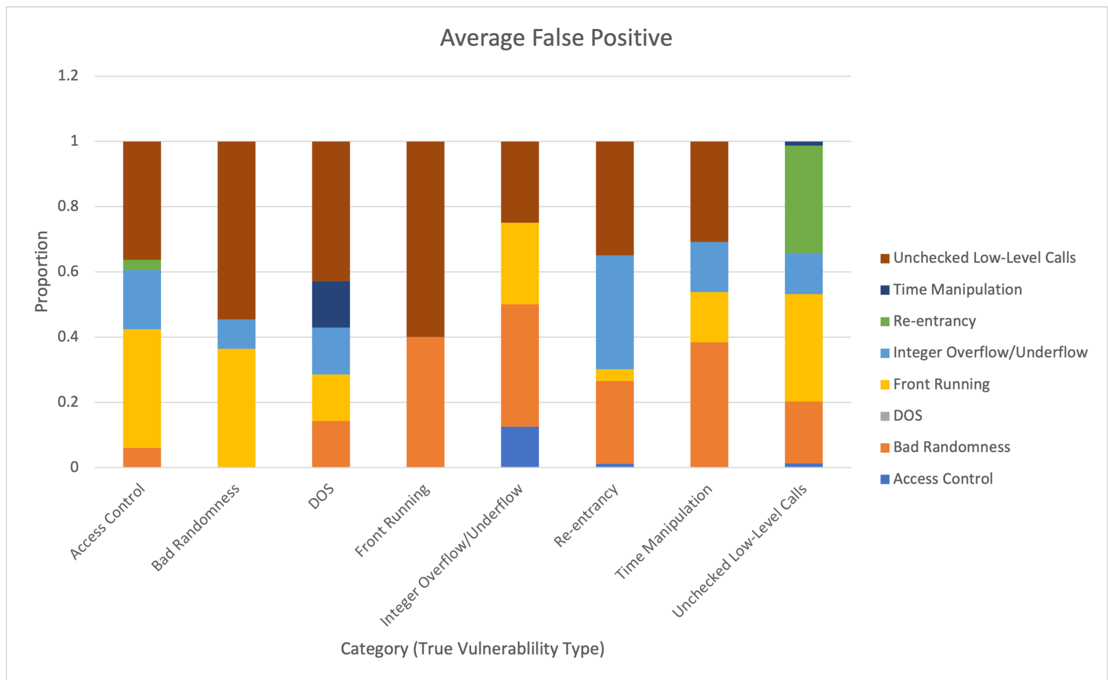
We first present our false positives. Recall, we define a false positive as a contract not containing a specific vulnerability but our specification detecting one. We calculate our false positives with the underlying assumption that each contract in $sb_{curated}$ only contains one type of vulnerability.



The horizontal groups represent contracts' original vulnerability categories (according to $sb_{curated}$); the vertical axis represents the number of false positive instances. For example, the contracts in the re-entrancy category get marked as having unchecked low-level calls 29 times [3]. According to the graph above, the primary contributors to false positives are unchecked low-level calls, integer overflow/underflow, and bad randomness. This can be attributed to the broadness of the specifications– contracts with Solidity functions

[3]Note: we are using multiple unchecked low-level call specifications on each re-entrancy contract.

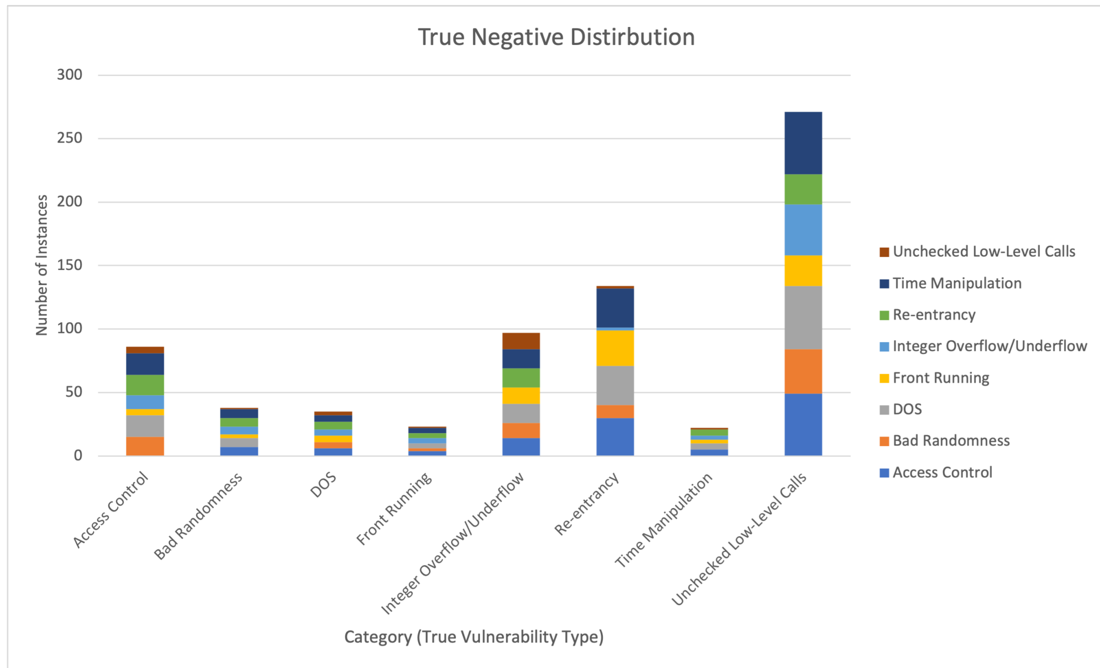
of `call`, `transfer`, `send`, etc. get marked for unchecked low-level calls, specific arithmetic operations get marked for integer overflow/underflow, and computations done using Solidity variables `block.timestamp`, `block.number`, and `blockhash` get marked for bad randomness [4]. The graph above is based on raw data, so it may seem that certain contract categories are more likely to be flagged as having other vulnerabilities. However, that is not the case– keep in mind that the categories have different numbers of contracts. The graph below proportionally shows the distribution of false positive by type in each category.



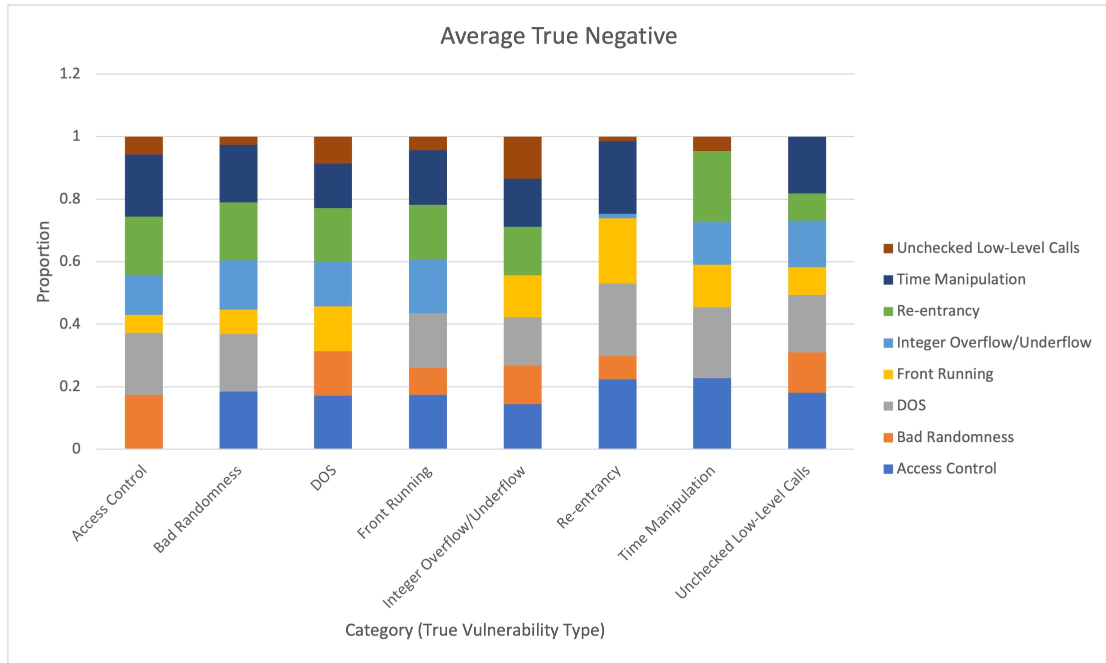
Going back to the example above: the re-entrancy category gets incorrectly marked 83 times. Re-entrancy contracts are identified as unchecked low-level calls 29 times. So, unchecked low-level calls make up roughly 35% of the re-entrancy false positives. Surprisingly, many of the contracts are being marked as having front running vulnerabilities. In the next steps of this research, we will inspect why this is.

We next present true negatives. Recall, a true negative occurs when a contract does not contain a specific vulnerability and our respective specification does not detect one.

[4] If we include the single opcode `TIMESTAMP` as a time manipulation specification, then we see several false positives in this category.



It is hard to see, but each column contains seven colors (corresponding to the vulnerabilities that are not the given category). For example, there are 86 true negatives in the access control category and 11 of these come from integer overflow/underflow. For the most part, the specifications correctly identify true negatives. However, the thinness of the maroon components corresponding to unchecked low-level calls shows that lack in identifying true negatives. This intuitively makes sense as this category accounts for several of the false positives found above. We present a different view of the true negatives akin to what was done in the false positive case.



This graph shows the proportion of true negatives in a category that were found from different vulnerability specifications. Going back to our example: roughly 12% true negatives in the access control category are accounted for by the integer overflow/underflow specification.

For the readers who prefer numbers over colored blocks, we give a table summarizing the data in the false positive and true negative categories. We abbreviate the specification names to save space.

False Positives

Vulnerability	AC	BR	DoS	FR	I O/U	R-e	TM	UC
Access Control	0	2	0	12	6	1	0	12
Bad Randomness	0	0	0	4	1	0	0	6
DoS	0	1	0	1	1	0	1	3
Front Running	0	2	0	0	0	0	0	3
Integer O/U	1	3	0	2	0	0	0	2
Re-entrancy	1	21	0	3	29	0	0	29
Time Manipulation	0	5	0	2	2	0	0	4
Unchecked Calls	1	15	0	26	10	26	1	0

The counterpart to this is:

True Negatives

	AC	BR	DoS	FR	I O/U	R-e	TM	UC
Access Control	0	15	17	5	11	16	17	5
Bad Randomness	7	0	7	3	6	7	7	1
DoS	6	5	0	5	5	6	5	3
Front Running	4	2	4	0	4	4	4	1
Integer O/U	14	12	15	13	0	15	15	13
Re-entrancy	30	10	31	28	2	0	31	2
Time Manipulation	5	0	5	3	3	5	0	1
Unchecked Calls	49	35	50	24	40	24	49	0

From the results, we need to improve our specifications. The DoS specifications are too limited, and do not cover all contracts exhibiting DoS. On the other hand, some specifications are too broad and find vulnerabilities in contracts not susceptible to that type of vulnerability, such as front running. More contracts, particularly in these two categories, need to be studied to find more vulnerability patterns and make the specifications more accurate. To do so, we can observe the contracts in sb_{wild}. We can, perhaps, use tools mentioned in Chapter 2 to categorize the contracts based on vulnerability susceptibility. Then use our methods on them.

5.2 Future Validation

The above approach is a starting point to corroborate our findings. But, it only checks for whether sequences of opcodes exist in the compiled code of the contracts. To address the concerns raised in the disclaimer discussion above and to address whether sequences are indeed reachable, further model-checking techniques need to be implemented. Below is an outline of a potential future validation method.

Our method will take, as input, a contract and vulnerability specification, and output the correctness of the specification. We first take a contract and create an SMV file from it. We can track the opcodes, changes to the stack, changes in gas balance, as well as other facets of the contract by creating respective SMV modules [5]. Our current LTL formulae

[5] In SMV, a *module* represents a finite state machine [26].

are based off of opcodes, but we can amend them to include other contract details. For example, we can monitor gas levels to detect Block Gas Limit DoS (mentioned in section 3.3). We then pass the generated SMV file and LTL specification into the nuXmv model checker. Either the specification is satisfied by the system or the model checker generates a counterexample. For example, the model checker might show a state is unreachable.

A more concrete example of a counterexample is the following. Recall our LTL formula for re-entrancy (section 4.6). We have

$$\phi_1 = PUSH1 \wedge \bigcirc(PUSH1 \wedge \bigcirc(MLOAD \wedge \bigcirc(DUP1 \wedge \bigcirc(DUP4 \wedge \bigcirc(SUB \wedge \bigcirc(DUP2 \wedge \bigcirc(DUP6 \wedge \bigcirc(DUP8 \wedge \bigcirc(GAS \wedge \bigcirc(CALL))))))))))$$

$$\phi_3 = SWAP3 \wedge \bigcirc(POP \wedge \bigcirc(POP \wedge \bigcirc(POP))).$$

We want $\phi_1 \wedge \bigcirc \neg \phi_3$ to show re-entrancy. Our model checker can yield a counterexample by presenting an execution trace which encapsulates operations changing `...PUSH1` → `PUSH1` → `MLOAD` → `DUP1` → `DUP4` → `SUB` → `DUP2` → `DUP6` → `DUP8` → `GAS` → `CALL` → `SWAP3` → `POP` → `POP` → `POP...` to show otherwise (a contract not being susceptible to re-entrancy).

We hope to better our specifications by incorporating more contract state and transition details, and have a more accurate validation method. Once we implement this method, we will test it on the sb_{wild} dataset. We will also compare it to some of the tools mentioned in Chapter 2.

Chapter 6

Conclusion

We have examined nine classes of vulnerabilities in Ethereum smart contracts. Using the categorization of contracts done by Durieux et al [31], we come up with LTL expressions that describe various types of vulnerabilities. We then present a preliminary validation [1] of the expressions against contracts. Overall, our specifications are accurate in identifying true positives; however, they can be improved for the access control and DoS categories. We find a high number of false positives– contracts flagged for certain vulnerabilities without necessarily displaying those. The findings warrant further pattern refinement which we discuss below along with our future plans.

6.1 Pattern Refinement

In this work, specifications are based off of contract execution traces. We, therefore, focus solely on EVM opcodes. There are some difficulties with this approach as mentioned in Chapter 4. The first is with trying to detect whether an execution trace is indeed a pattern or simply contract-specific. This analysis can be done, perhaps, by regarding the control flow graphs of contracts and inferring patterns in execution routes. This will help us differentiate between which instructions are critical to a vulnerability pattern and which ones are used in intermediate computations of contracts (regardless of association to vulnerability).

[1] Our validation excludes the short address category.

On the other hand, there is an issue of the pattern being too broad. Take, for instance, integer overflow/underflow. Currently, the specification searches for general arithmetic operations. But this results in several contracts being flagged for this vulnerability without actually being vulnerable. A further level of specificity in vulnerability expressions is needed to reduce the number of false positives. One way to do so is by adding another component to our specification– such as tracking changes to the stack. Let us say we want to detect division by zero [2] instead of just division in general. According to the Yellow Paper [51], the integer division operation, `DIV`, checks if the second item of the stack is 0. If so, it removes the first two items of the stack, and adds an item with value 0. If not, the added item is the floor of the result of dividing the two. Therefore, if we track the value of the second item on the stack before a `DIV` operation, and see that it is 0, we can detect an integer overflow/underflow error more precisely. Likewise, we can track additional changes such as a contract's gas balance (which would be useful in DoS for example).

6.2 Further Validation

Our next validation steps are two-fold. First, we will implement our NuSMV-based model checking approach. As mentioned in Chapter 5, this will help us refine our search for vulnerabilities. We will be able to check whether the flagged points are indeed reachable or not and whether there is a possibility of attack; thereby, reduce the number of false positives. Next, we will test our methods on the sb_{wild} dataset. There may not be a way to corroborate our findings (unless we run other tools on this set), but it is will be interesting to test our specifications on more complicated real-world data.

6.3 Future Inspiration

We also hope to expand our work to cover more types of vulnerabilities in smart contracts. As mentioned in Chapter 4, some of formulations do not fully span the given category. For example, overflows in shift operations and signed to unsigned conversions are not included in the integer overflow/underflow formula. There are other types of vulnerabilities outside the DASP10 categories. As Ethereum transactions often involve the exchange of Ether,

[2] This is moot as Solidity supports a catch clause to handle this exception.

creating specifications relating to gas-based vulnerabilities (such as wallet griefing) are of interest.

We further would like to observe contracts written in other languages such as Vyper. A hypothesis that is yet to be tested is do the specifications found in this work apply to vulnerabilities in contracts written in other languages. EVM natively executes low-level bytecode, so contracts written in high-level languages need to be translated to this lower level. The specifications in this work are based off of low-level code– can it be inferred that patterns manifest regardless of the languages contracts are written in.

This work is inspired by the question of whether we can encapsulate vulnerabilities in Ethereum smart contracts as specifications written in linear temporal logic. We have done so for certain classes of vulnerabilities such as re-entrancy. But further work needs to be done to improve upon our specifications, to test them on a larger set of contracts, and to find specifications for other types of vulnerabilities. We look forward to continuing our work.

References

[1] Bitcoin. https://bitcoin.org/en/.

[2] C library function - memcpy(). https://www.tutorialspoint.com/c_standard_library/c_function_memcpy.htm.

[3] Decentralized security project. https://dasp.co.

[4] EthBMC. https://github.com/RUB-SysSec/EthBMC.

[5] Etherscan. https://etherscan.io.

[6] eThor. https://secpriv.wien/ethor/.

[7] HoRSt. https://secpriv.wien/horst/.

[8] How the winner got fomo3d prize — A Detailed Explanation.

[9] Lock (computer science). https://en.wikipedia.org/wiki/Lock_(computer_science).

[10] MadMax. https://github.com/nevillegrech/MadMax.

[11] Manticore. https://github.com/trailofbits/manticore.

[12] Securify. https://github.com/eth-sri/securify/.

[13] SWC Registry: Smart Contract Weakness Classification and Test Cases. https://swcregistry.io.

[14] teEther. https://github.com/nescio007/teether.

[15] Vandal. https://github.com/usyd-blockchain/vandal.

[16] Vyper. https://vyper.readthedocs.io/en/stable/.

[17] Ethereum Smart Contract Best Practices. https://consensys.github.io/smart-contract-best-practices/, November 2008.

[18] teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Proceedings of the 27th USENIX Security Symposium*. USENIX, August 2018. https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-krupp.pdf.

[19] py-solc-x. https://solcx.readthedocs.io/en/latest/, 2020. Revision 26e1a78a.

[20] Ethereum Whitepaper. https://ethereum.org/en/whitepaper, October 2021.

[21] Remix. https://remix.ethereum.org, 2021. Version 0.19.0.

[22] Rajeev Alur. *Principles of Cyber-Physical Systems*. MIT Press, 2015.

[23] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A Survey of Attacks on Ethereum Smart Contracts. In *Proceedings of the 6th International Conference on Principles of Security and Trust*, volume 10204. Association for Computing Machinery Digital Library, April 2017. https://dl.acm.org/doi/10.1007/978-3-662-54455-6_8.

[24] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A Scalable Security Analysis Framework for Smart Contracts. September 2018. https://arxiv.org/pdf/1809.03981.pdf.

[25] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv Symbolic Model Checker. In *CAV*, pages 334–342, 2014. http://dx.doi.org/10.1007/978-3-319-08867-9_22.

[26] Roberto Cavada, Alessandro Cimatti, Charles Arthur Jochim, Gavin Keighren, Emanuele Olivetti, Marco Pistore, Marco Roveri, and Andrei Tchaltsev. *NuSMV 2.6 User Manual*. FBK-irst, 2010. https://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf.

[27] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1 edition, 1999.

[28] Edmund Clarke, Thomas Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer International Publishing, 2018.

[29] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. SB Curated: A Curated Dataset of Vulnerable Solidity Smart Contracts. https://github.com/smartbugs/smartbugs/tree/master/dataset.

[30] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. SmartBugs Wild Dataset. https://github.com/smartbugs/smartbugs-wild.

[31] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 530–541. Association for Computing Machinery Digital Library, June 2020. https://dl.acm.org/doi/10.1145/3377811.3380364.

[32] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. Sok: Transparent dishonesty: Front-running attacks on blockchain. In *Lecture Notes in Computer Science*, volume 11599, pages 170–189. Springer, March 2020. https://link.springer.com/chapter/10.1007/978-3-030-43725-1_13#citeas.

[33] Ethereum. *Solidity Read the Docs*, v0.8.10 edition, 2016-2021. https://docs.soliditylang.org/en/v0.8.10/index.html#.

[34] Joel Frank, Cornelius Aschermann, and Thorsten Holz. EthBMC: A Bounded Model Checker for Smart Contracts, August 2020. https://www.usenix.org/system/files/sec20-frank.pdf.

[35] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts. In *Proceedings of the ACM on Programming Languages*, volume 2. Association for Computing Machinery Digital Library, November 2018. https://dl.acm.org/doi/pdf/10.1145/3276486.

[36] Yoichi Hirai. Exception on Overflow. https://github.com/ethereum/solidity/issues/796#issuecomment-253578925.

[37] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing Safety of Smart Contracts. In *Network and Distributed Systems Security (NDSS) Symposium*, pages 18–21, February 2018. http://pages.cpsc.ucalgary.ca/~joel.reardon/blockchain/readings/ndss2018_09-1_Kalra_paper.pdf.

[38] Saul Kripke. *A Completeness Theorem in Modal Logic*, volume 24, pages 1–14. Association for Symbolic Logic, 1 edition, March 1959. https://www.jstor.org/stable/2964568.

[39] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 254–269. Association for Computing Machinery Digital Library, October 2016. http://dx.doi.org/10.1145/2976749.2978309.

[40] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1186–1189. IEEE, November 2019. https://ieeexplore.ieee.org/document/8952204.

[41] Bernhard Mueller. Smashing Ethereum Smart Contracts for Fun and Real Profit. 2018. https://github.com/b-mueller/smashing-smart-contracts/blob/master/smashing-smart-contracts-1of1.pdf.

[42] Tasuku Nakamura. Solidity By Example. https://solidity-by-example.org, 2018.

[43] Ivica Nikolic. Maian. https://github.com/ivicanikolicsg/MAIAN.

[44] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 653–663. Association for Computing Machinery Digital Library, December 2018. https://dl.acm.org/doi/10.1145/3274694.3274743.

[45] Nikhil Parasaram. Mythril. https://github.com/ConsenSys/mythril.

[46] Eric Rafaloff. Analyzing the ERC20 Short Address Attack. https://ericrafaloff.com/analyzing-the-erc20-short-address-attack/.

[47] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffe. eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 621–640. Association for Computing Machinery Digital Library, October 2020. https://dl.acm.org/doi/10.1145/3372297.3417250.

[48] David Siegel. Understanding the dao attack. https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/.

[49] Facu Spagnuolo. Ethereum in Depth. https://blog.openzeppelin.com/ethereum-in-depth-part-2-6339cf6bddb9/.

[50] Petar Tsankov, Andrei Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 67–82. Association for Computing Machinery Digital Library, October 2018. https://dl.acm.org/doi/10.1145/3243734.3243780.

[51] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger. Technical Report Version fa00ff1, Ethereum & Parity, 2015.

[52] Xiao Liang Yu. Oyente: An Analysis Tool for Smart Contracts. https://github.com/melonproject/oyente.