Testing Smart Contracts

by

Alexander Leid

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science (Computer Science) in the Faculty of Science at Stellenbosch University

Supervisor: Prof. AB van der Merwe
Co-supervisor: Prof. W Visser

March 2020

The financial assistance of the Council for Scientific and Industrial Research (CSIR) is hereby acknowledged.

Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: March 2020 .....

Abstract

Testing Smart Contracts

A. Leid

Computer Science Division,
Department of Mathematical Sciences,
University of Stellenbosch,
Private Bag X1, 7602 Matieland, South Africa.

Thesis: MSc (Computer Science)

March 2020

There have been several high-profile exploits of smart contracts running on the Ethereum Virtual Machine (EVM) over the last few years since the re-lease of Ethereum. Many of these exploits were introduced via programmer error and could be avoided by proper auditing beforehand. Security analysis tooling has advanced in this space to aid developers and auditors to auto-matically find these exploits and in some cases generate test input that can recreate the exploit. In this work, we review the most critical vulnerabili-ties currently present in the EVM ecosystem and provide best practices and forms of prevention. Taxonomies (new and existing) are presented to cate-gorise the type of smart contract exploits present at the application layer and compare them to similar exploits in imperative programs. Automated testing tools are investigated and extended in areas where they may struggle to detect certain vulnerabilities and to synthesise adversarial smart contracts. Lastly, some of the most popular and actively developed automated testing tools are catalogued, evaluated, and benchmarked.

Acknowledgements

First, I would like to thank my supervisors, Prof. Brink van der Merwe and Prof. Willem Visser, for their expert insight, assistance, and guidance over the course of my studies.

I would like to acknowledge the financial support of the CSIR for this work and hope that they can continue enabling more students in this country to pursue their postgraduate studies like I have.

Lastly, I would like to thank my family and friends for their unwavering support throughout all my years of study and especially while preparing this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

After the early success of Bitcoin [1], the first major cryptocurrency, some members of its community started to investigate ways of running increasingly complex user applications on a public, decentralised blockchain. One of the first and most popular cryptocurrency platforms to focus specifically on this trait is Ethereum.

Ethereum is a distributed computational platform that allows users to run programs known as smart contracts, first conceived by Nick Szabo in the 1990s [2], that can autonomously respond to human blockchain interactions according to a programmed set of rules [3]. Unlike Bitcoin, the scripting instruction set available at runtime in Ethereum is designed to be Turing-complete in theory, but practically limited by a maximum computational allowance.

One key aspect of the Ethereum platform is that once a contract is deployed by the user, the contract's instruction code will not be able to be altered any further. Naturally this immutability can spell disaster when combined with program bugs originating from user error and the handling of monetarily significant cryptocurrency assets.

The most infamous example of a smart contract bug causing extreme loss of monetary value is that of The DAO, a Decentralised Autonomous Organisation (DAO) designed to offer decentralised governance of enterprise organisations [4]. As a result of an overlooked software bug, a single user was able to steal roughly 3.6 million Ether [5], which at the time was worth approximately $50 million.

Vulnerabilities like that of The DAO hack do not exploit any bugs or issues in the platform itself, but instead rely on unintentional, yet programmatically valid behaviour written by the smart contract developers. With this large amount of room for error, the onus is on the developers to ensure that their contracts are sufficiently tested and audited for vulnerabilities before deployment.

1.1 Testing Smart Contracts

A smart contract developer has many options at their disposal to ensure that a contract is sufficiently tested for vulnerabilities before it is permanently deployed to the blockchain.

Generally the first method of testing smart contracts would be to write unit and integration tests. Solidity, a popular high-level smart contract language targeting the Ethereum runtime environment [6], already has a mature testing framework in the form of Truffle [7] that is well supported by its community and allows developers to test against a local test Ethereum network.

There are, however, an increasing number of automatic testing and security analysis tools that are available to smart contract developers and auditors. These tools can be used to quickly cover large amounts of code and find bugs, like that of The DAO hack, without any user input or guidance.

In this work we first investigate and survey the common vulnerabilities that exist in the Ethereum application layer. With a computational environment like Ethereum that is constantly being developed, it is common that some software vulnerabilities may be phased out with new updates. Although it is impossible to catalogue each and every possible vulnerability, since one application's bug may be another's feature, we can at least attempt to narrow down the ones that are generally considered to be an issue.

Next we survey the security analysis tools at our disposal and note the types of vulnerabilities that each claims to identify. We take the time to investigate the use of these tools and also extend one of them, the Manticore symbolic execution tool [8], thus demonstrating how it can be used during the auditing process to find bugs that are extremely difficult to catch with manual inspection.

Lastly, we evaluate these tools to determine how effective they are in automatically detecting certain vulnerabilities and how efficient they are in covering the code of larger contracts.

1.2 Contributions

The following contributions are made as part of this work:

1. Cataloguing the vulnerabilities and security analysis tools present in the Ethereum ecosystem:
    - Identification of the most severe smart contract vulnerabilities and methods of preventing them;
    - Identification of the most popular automated security analysis tools;
    - Taxonomisation of vulnerabilities based on common grouping criteria; and

- Taxonomisation of tooling according to vulnerability classification.

2. Investigating the use of the Manticore symbolic analysis tool in our own framework, Manticore+:
   - Modelling key elements of the Ethereum ecosystem symbolically;
   - Automatically detecting a variety of vulnerabilities; and
   - Synthesising attacking smart contracts used as part of exploit sequences.
3. Evaluating the effectiveness and efficiency of the investigated security analysis tools:
   - Comparing their effectiveness when automatically detecting vulnerabilities;
   - Analysing their ability to reach invariant states; and
   - Investigating performance and efficiency when applied to contracts from the main Ethereum network.

1.3 Overview

Following this introductory chapter, Chapter 2 provides the reader with background information on Ethereum, the Solidity programming language, fuzz testing tools and symbolic execution. Chapter 3 then explains each investigated Ethereum vulnerability in-depth (with examples) and demonstrates prevention methods and best practices to avoid them. Then we present a list of state-of-the-art security analysis tools in Chapter 4 that is able to automatically detect these vulnerabilities. Next, Chapter 5 demonstrates how one of these tools, Manticore, can be used as part of a framework and extended to perform advanced security analysis and exploit synthesis. We then evaluate all these tools in terms of their efficiency and effectiveness in Chapter 6, before finally concluding the thesis in Chapter 7.

Chapter 2

Background

This chapter serves to give the reader a brief overview of the Ethereum platform and its most popular programming language Solidity, as well as an explanation of the testing and verification methodologies presented later in this thesis.

Vulnerabilities specific to Ethereum and its smart contracts will be discussed separately in Chapter 3, but will assume a basic understanding of the system as presented in this chapter. Tool-related discussions will only appear afterwards in Chapter 4 to keep the background sections tool-agnostic.

2.1 Ethereum

Ethereum is a distributed computing platform hosted on a public, open-source blockchain initially proposed by Vitalik Buterin in a white paper written in 2013 [3]. This platform allows users to run programs, known as smart contracts, within an environment similar to the Bitcoin blockchain [1].

The key difference is that the scripting language for Bitcoin transactions, known as Script [9], is intentionally designed to not be Turing-complete or to keep track of an inter-transactional state and can thus not be used to run the complex user applications intended to run on Ethereum.

2.1.1 Release Management

As with many other cryptocurrency platforms, Ethereum is constantly developing and must always be ready to respond to new challenges and feedback from its community. Some of the more significant system milestones such as changes to the consensus protocol were planned from the start, while other modifications to the standard may be proposed by users via an Ethereum Improvement Proposal (EIP). Table 2.1 lists some of the major upgrades that implement various milestones and EIPs as part of a hard fork [10].

In the past the Ethereum community has also responded to major security breaches by implementing forks in the codebase and blockchain. One such example is the hard fork implemented after the DAO hack, allowing users to retrieve their funds from a curator contract [11], which subsequently also spawned the Ethereum Classic platform as a continuation of the original blockchain.

For the purposes of this thesis, we are only concerned with the impact that major releases have on the vulnerabilities discussed and whether or not the testing and verification tools are compatible with the latest version. Testing and symbolic execution frameworks that emulate the EVM are generally not affected by changes related to mining and consensus protocols, but must be brought up to date with changes directly affecting program execution (e.g. adding additional opcodes or changing the way fees are handled).

#	Date	Name
0	2015-07-30	Frontier
1	2015-09-07	Frontier Thawing
2	2016-03-14	Homestead
3	2016-07-20	After DAO attack
4	2016-10-18	Tangerine Whistle
5	2016-11-22	Spurious Dragon
6	2017-10-16	Byzantium
7	2019-02-28	Constantinople

Table 2.1: List of major Ethereum forks.

2.1.2 Ether

Ethereum also features a tradeable cryptocurrency token called Ether. Each account, whether owned by a user or operating as a smart contract, may have a balance of Ether and trade it freely with other accounts. Developers may also program additional behaviour in their smart contracts that acts on incoming transactions that send Ether.

Ether doubles as a currency used to pay for computation performed by smart contracts, known as the gas fee, whereby the cost of each transaction is relative to the total cost of each instruction executed by the EVM.

The currency has a zero decimal precision, with a single base unit referred to as Wei. This is typically the unit of measurement when denoting transaction fees. Since these fees are orders of magnitude smaller than the amount of Wei mined per transaction, additional denominations were introduced as per the Ethereum white paper [3]:

10^0**:** Wei

10^{12}**:** Szabo

10^{15}**:** Finney

10^{18}**:** Ether

Amounts transferred between parties are therefore also generally denoted in Ether.

2.1.3 Accounts and State

The world state in the context of Ethereum is described as a mapping between 20-byte addresses, known as accounts, and their account states [12]. The account state is defined as follows:

- A nonce value indicating the number of transactions signed or contracts instantiated by the account that increments each time;
- The Ether balance of the account in Wei;
- A 256-bit hash pointing to the account's internal state storage; and
- An immutable hash of the EVM code of the account, if applicable.

These accounts are generally grouped into:

- User accounts that are controlled by a private key and possess no EVM code of their own; and
- Contract accounts that contain code and act autonomously as defined by this program.

The account nonce value ensures that users can not "double-spend" by submitting more than one transaction at the same time, since the nonce value will increment between calls and therefore treat them as distinct messages.

2.1.4 Transactions

The world state of the Ethereum blockchain is modified by the application of a successful transaction. Transactions are grouped into blocks which are mined on average approximately every 12 seconds, as determined by the Ethereum network's current difficulty [13]. All fees associated with the block are paid by the accounts submitting the transactions and rewarded to the account address of the miner that successfully mined the block.

Ethereum transactions are specified to have the following parameters:

- Fields representing the sender's cryptographic signature;
- The nonce value of the sender;
- The recipient's 20-byte address (empty during contract initialisation);
- The amount of Ether paid per unit of gas spent during execution (expressed in Wei);
- The maximum number of gas units the sender is willing to pay for;
- The amount of Ether sent with the transaction; and
- A byte array of unlimited size representing either optional call data or the code of the newly created contract during contract initialisation.

Each instruction executed within the EVM and additional bytes of call data costs a predetermined amount of gas, which is subtracted from the initial starting gas value sent above. This amount of gas is then multiplied by the gas price value specified in the transaction and paid upfront by the sender to the miner. If execution halts gracefully, then all remaining gas is refunded to the sender. Otherwise, the world state may be reverted without refunding any gas to the sender. Ethereum also features a global gas limit per block as voted on by miners.

2.1.5 Messages

Contracts within the Ethereum platform are able to send messages to other contracts, similar to the transactions described earlier in Section 2.1.4. These messages act like remote-process calls, but with subtle differences in the way each contract's storage is managed during execution. A call consists of the following elements similar to a transaction:

- The sender of the message;
- The recipient;
- An Ether amount transferred;
- A data byte-array;
- Available gas; and
- The gas price.

Additionally, these messages contain:

- The sender of the initial block-level transaction (or originator);

- The address of the account that contains the code that must be executed;
- The depth of the message call stack; and
- A flag to allow modifications to state.

These messages will return a byte-array value as output and behave similar to transactions in terms of gas usage.

One important thing to note is that any changes made to the contract's storage preceding message calls will remain once the call is made. In other words, the world state enters a sub-state during message execution and care must be taken by the contract developer to ensure that it is not left in an invalid state. The contract might be the recipient of another recursive message further down the line if enough gas was passed to the recipient, which was the basis for the infamous DAO attack.

2.1.6 Pre-compiled Contracts

There are a number of reserved addresses in Ethereum that implement frequently used cryptographic functions that are callable by other contracts. The number of these so-called pre-compiled contracts have increased with time and are currently up to eight, as of the Byzantium release. The address and name of the function is listed in Table 2.2, with a more technical description given in the Yellow Paper [12].

Address	Function
0x1	ECDSA signature recovery
0x2	SHA256 hash
0x3	RIPEMD160 hash
0x4	Identity
0x5	Modular exponentiation
0x6	Elliptic curve addition
0x7	Elliptic curve scalar multiplication
0x8	Elliptic curve pairing check

Table 2.2: List of pre-compiled contracts as of Byzantium.

2.1.7 The Ethereum Virtual Machine

The EVM is a stack-based virtual machine with a 256-bit word size and maximum stack depth of 1024. This specific word size was chosen because of its use in various cryptographic functions often performed by the EVM.

Control Flow

Control flow can be terminated with any of the instructions shown in Table 2.3, along with the added side-effects of removing the contract from the blockchain and transferring its Ether to the specified address in the case of a `SELFDESTRUCT`. Any other opcode not defined in the specification will also terminate execution of the current message, but not refund the remaining gas to the caller.

Hex Code	Mnemonic	Description
00	STOP	Stop execution
F3	RETURN	Return a value to the current message caller
FD	REVERT	Revert execution and returns the specified value from memory
FF	SELFDESTRUCT	Terminates the current contract and transfers its Ether balance to the specified address

Table 2.3: List of EVM opcodes related to control flow.

Arithmetic Instructions

The arithmetic and mathematical operations included in Table 2.4 are all returned modulo 2^{256} to the stack. High-level languages targeting the EVM, such as Solidity, are able to further manipulate these values using additional instructions and truncate bits to effectively work with smaller word sizes. Division by zero will return zero as per the yellow paper specification, but higher-level languages generally halt execution in such cases.

Hex Code	Mnemonic	Description
01	ADD	Adds the top two elements on the stack, modulo 2^{256}
02	MUL	Multiply the top two elements on the stack, modulo 2^{256}
03	SUB	Subtracts the top two elements on the stack, modulo 2^{256}
04	DIV	Division of the top two elements on the stack
05	SDIV	Unsigned division of the top two elements on the stack
0A	EXP	Unsigned exponentiation of the top two elements on the stack
20	SHA3	Keccak256 hash of the value at the specified memory location

Table 2.4: List of EVM opcodes related to arithmetic and mathematical operations.

Environmental Variables

Table 2.5 notes the opcodes related to some of the EVM-specific features, such as information on the current message parameters or gas quotas.

Hex Code	Mnemonic	Description
30	ADDRESS	Address of the current contract
31	BALANCE	Ether balance of the specified address in Wei
32	ORIGIN	Address of the transaction originator
33	CALLER	Address of the message sender
34	CALLVALUE	Ether value of the message in Wei
3A	GASPRICE	Ether price per unit of gas for the current message in Wei
40	BLOCKHASH	Hash of the specified block
41	COINBASE	Address of the current transaction fee beneficiary
42	TIMESTAMP	Timestamp of the current block
43	NUMBER	Current block number
44	DIFFICULTY	Current block difficulty
45	GASLIMIT	Block gas limit
5A	GAS	Amount of remaining gas

Table 2.5: List of EVM opcodes related to environmental and call data.

Storage and Memory

The EVM supports both a volatile memory space and a storage area of fixed size that persists between calls, each able to index up to 2^{256} elements. Reading and writing opcodes for memory and storage are both shown in Table 2.6.

Hex Code	Mnemonic	Description
51	`MLOAD`	Load the 256 bit word at the specified offset from memory
52	`MSTORE`	Write a 256 bit word to the specified offset in memory
54	`SLOAD`	Read the 256 bit word at the specified offset from storage
55	`SSTORE`	Write a 256 bit word to the specified offset in storage

Table 2.6: List of EVM opcodes related to storage and memory functions.

Messaging

Table 2.7 displays the EVM opcodes that initialise new contracts and send calls. The key difference between the standard `CREATE` and `CREATE2` opcodes is that the newly created contract's address can be determined beforehand in the latter case. Under normal circumstances, the new contract's address is calculated based off the current address and nonce, whereas the second method bases it off of the current address and initialisation code being passed to the contract. External applications therefore do not have to rely on a specific nonce value to calculate new addresses beforehand.

The message calling opcodes (`CALL`, `DELEGATECALL` and `STATICCALL`) are mainly differentiated by their handling of contract storage, as described in Table 2.7.

Hex Code	Mnemonic	Description
F0	CREATE	Create a new contract with the code specified in memory
F1	CALL	Send a message to another contract
F4	DELEGATECALL	Send a message to another contract with access to the current contract's storage
F0	CREATE2	Create a new contract with the code specified in memory and set its address to a deterministic value
FA	STATICCALL	Send a message to another contract without allowing any writes to storage

Table 2.7: List of EVM opcodes related to messaging.

2.2 Solidity

With the EVM as reference specification, developers are able to design high level languages that compile down to EVM bytecode. One early example of this was a simple, low-level language called LLL [14] (short for Lisp-Like Language) that produced very small binaries. At the same time a high-level language called Solidity [6] was developed officially by the Ethereum team and soon became the most popular language for implementing smart contracts.

This section will serve to both introduce the reader to some of the basic syntax necessary to comprehend our example code and to highlight features of the language that may expose critical vulnerabilities. While there are other high-level languages that target the EVM, such as Vyper [15] that emphasise ease of verification and security, we will rather be focusing on Solidity due to its popularity.

2.2.1 Language Properties

Solidity is a statically-typed, object-oriented language that is used to write smart contracts for the EVM. Contracts are implemented as objects, where inter-contract method calls spawn new Ethereum messages. Developers are also able to make use of inheritance, libraries, user-defined structs and all the various EVM environment variables discussed in Section 2.1.7.

2.2.2 Basic Syntax

For this section we use Listing 1 to explain some of the fundamental elements of a Solidity program's syntax, before moving on to the intricacies of the language in later sections. In Solidity, the keywords, `contract` and `function`,

are reserved to write an object-like contract and its methods (bounded by curly brackets), which in our example is a contract called "Basic" with one method called "test". The contract also contains a function that will be executed only when the contract is created, specified by the keyword: `constructor`. This constructor function may be left out to simply initialise a contract without modifying any of its storage or it can be used to perform other actions as in our example. Just above our constructor we declare a variable named "owner"

```
1  pragma solidity ^0.5.10;
2
3  contract Basic {
4      address owner;
5
6      constructor () public payable {
7          // Optional constructor function
8          owner = msg.sender;
9      }
10
11     function test() public payable {
12         require(msg.sender == owner);
13     }
14 }
```

Listing 1: Basic syntax example.

that is of type:`address`. Variables that are declared outside functions denote persistent storage variables within the EVM and will by default be initialised to a zero value. This `address` variable specifies the address of an account within Ethereum and can be used to, for example, send messages or read balances.

Back in the constructor we first assign the owner during creation by reading the `msg.sender` environmental variable. The built-in `msg` refers to the current message and the `sender` member will return the address of the current message sender (i.e. it will call the `CALLER` opcode).

Our only statement in the `test()` function calls the built-in `require` function that takes one boolean argument and reverts execution if it evaluates to `false`. Note that our test function also has two modifiers: `public` and `payable`. The following built-in function and variable modifiers are available:

private Only visible in the current contract and no derived ones

internal Only visible inside the current contract or ones that derive it

public Visible inside and outside the current contract

external Only callable by initiating a transaction or message

payable Indicates that the function or constructor may receive Ether when called, whereas the default behaviour for a function without this modifier is to revert immediately if a non-zero Ether value is sent.

The command on the first line, `pragma solidity ^0.5.10;`, simply enforces the specified compiler version. A `^` symbol before the version indicates that all minor releases after the specified version are also allowed, however, this is considered to be bad practice, since the compilation output may no longer be the same as the bytecode that was tested.

2.2.3 Types

In the previous section we mentioned the `address` type as part of our basic example, but we will now briefly highlight some of the other types offered by Solidity.

Integers

Integers in Solidity are available both in signed (`int`) and unsigned (`uint`) forms, with signed integers represented in two's complement form as per the EVM specification. The default byte size for integers is 256 bits, which is again the same as the EVM word size. If necessary, a developer may declare variables with smaller sizes in multiples of 8 up to 256 (e.g. `uint8`, `int8`).

Booleans

Constant boolean values in Solidity are denoted by the `true` and `false` keywords, with support for the following logical operators:

- equality: `==`
- inequality: `!=`
- conjunction: `&&`
- disjunction: `||`
- negation: `!`

Addresses

In our example contract we assigned the 20-byte sender address of the initial contract creation to the storage variable of type `address`. In addition to this, Solidity version 0.5.0 added a new type: `address payable`, which ensures that the compiler will only allow these types of addresses to receive funds. Furthermore, `address payable` variables may be freely casted to the regular `address` type, but not vice versa. Objects of both of these address types have access to the `balance` property, which will invoke the low-level `BALANCE` opcode on that address and return its Ether balance in Wei. In Section 2.2.6 we will further discuss the list of message sending methods available only to `address payable` objects.

Contracts

User-defined contract classes, like the one defined in our earlier example, act as extensions of the `address` type and may be used to:

- Create new instances on the blockchain of that contract;
- Instantiate an existing contract account;
- Call functions on contract instances;
- Read properties; and
- Perform the built-in functions provided by the `address` base class.

Contracts may be created using the keyword: `new`, and assigned to an instance variable using

```
Contract c = new Contract();
```

or instantiated from existing accounts using their addresses (represented here by an `address` literal):

```
Contract c = Contract(0xBf4eD7b27F1d666546E30D74d50d173d20bca754);
```

Arrays

Arrays in Solidity are available in fixed or dynamic length formats, declared to be used for unsigned integers, for example, as `uint256[42]` or `uint256[]`, respectively. An array's length may be retrieved using the `length` property and values may be appended or removed using the `push()` and `pop()` methods. The `byte` and `string` arrays are also available as more tightly packed versions of the `byte[]` array, since it does not pad every byte to 32-byte words.

Mappings

Hash maps are implemented as "mappings" in Solidity and may be declared with keys of type `K` and values of type `V` using:

```
mapping(K => V)
```

The key type may not be a contract, array or other mapping, whereas the value type has no restrictions.

2.2.4 Functions

We will use this section to explain the way that contract functions are structured within the EVM bytecode generated by the Solidity compiler. These functions not only have to be callable from within the same contract instance, but they also have to adhere to a universal format of message calling either originating from other contracts or user accounts. This message format is defined as the Application Binary Interface (ABI) and it specifies how a given Solidity function may be called with the byte array data parameter.

When a Solidity contract receives a message or transaction, it will analyse the first four bytes in the message's data string for a function selector. If the data string contains a valid selector that corresponds to one of the contract's available functions, then the program counter will jump to that function's code location. The function selector is simply the first 4 bytes of the Keccak-256 hash of the function signature, which is the function name followed by the types of all its parameters in parenthesis.

In the event that the identifier does not match any valid options or if the data string is empty, the contract will execute what is called the fallback function. The fallback function is always present, but may be overridden in solidity by defining a nameless function. Also note that the fallback function must be marked payable for it to receive funds like other functions. The only way to avoid this and still receive funds when a contract's functions are not marked as `payable`, is for the contract to be marked as the beneficiary of a coinbase transaction (i.e. the recipient of all a block's gas fees) or a self-destruct.

To demonstrate an example of the function selector, Listing 2 contains a small sample program that acts as a user's personal wallet stored on the blockchain with the following four functions:

- The `constructor`;
- `withdraw()`;
- `transferTo(address,uint256)`; and
- The fallback function defined using a nameless function.

Our compiled bytecode will generate a function selector for both the `withdraw()` and `transferTo(address,uint256)` function signatures using the following Keccak-256 hashes:

$$\text{keccak256}(\text{“withdraw()”}) = \texttt{0x3CCFD60B2E3DDCE51AB210B}\ldots$$
$$\text{keccak256}(\text{“transferTo(address,uint256)”}) = \texttt{0x2CCB1B30FAB9B61DF5BAC0D}\ldots$$

```
1  pragma solidity ^0.5.10;
2
3  contract Wallet {
4      address payable owner;
5
6      constructor () public {
7          owner = msg.sender;
8      }
9
10     function withdraw() external  {
11         if (msg.sender == owner) {
12             owner.transfer(address(this).balance);
13         }
14     }
15
16     function transferTo(address payable recipient,
17                         uint256 amount) external {
18         if (msg.sender == owner) {
19             recipient.transfer(amount);
20         }
21     }
22
23     function () external payable {
24         // This default function may receive Ether
25     }
26 }
```

Listing 2: Simple wallet contract.

2.2.5 Storage Access

The EVM allows each contract account to access a state storage area that maps a 32-byte address space to 32-byte words. Storage variables persist between successful transactions and also between message calls, unless reverted.

Solidity starts allocating storage by placing each state variable sequentially beginning at offset 0 in the order that they are declared in. Any types that are smaller than the 256-bit word size will be padded and, if possible, packed in with other adjacent smaller variables. In the example contract

```
contract Example {
    uint256 a;
    uint256 b;
    uint128 c;
    uint128 d;
}
```

we have four state variables that will be packed as follows:

Offset	256-bit word	
0	`uint256 a`	
1	`uint256 b`	
2	`uint128 c`	`uint128 d`

Alternatively, if we had declared the variables in a different order as in

```
contract Example {
    uint256 a;
    uint128 b;
    uint256 c;
    uint128 d;
}
```

then both the `uint128` variables would be padded to 256 bits and occupy the following storage configuration:

Offset	256-bit word
0	`uint256 a`
1	`uint128 b`
2	`uint256 c`
3	`uint128 d`

While both approaches semantically describe the same program, the first configuration would save gas on storage operations, since both variables could be written at the same time.

Array Access

Arrays defined outside functions, as persisted state variables, have their values stored in the same address space as all other state variables. The length of the

array is stored as a 256-bit integer at the offset equal to the position that it was declared at in the source code (with packing accounted for other variables).

Array values, on the other hand, are stored further away determined by the hash value of the array's sequential offset. Given an element's index i for an array declared at offset n, the final offset in storage of the value will be at position

$$\text{keccak256}(n) + i.$$

Note that if an index value is passed such that the offset is greater then or equal than 2^{256}, the offset value will overflow and wrap around to start at 0 again.

2.2.6 Built-in Functions and Variables

In this section we highlight several important built-in functions and variables provided by Solidity that will be of use when explaining certain vulnerabilities and tools in later chapters.

Environmental Data

Solidity provides functions and global variables that correspond to the low-level environmental EVM instructions discussed in Section 2.1.7. Three important objects are made available to each function:

- `block` for properties related to the current block being mined;
- `msg` containing data related to the current message (regardless of message depth); and
- `tx` for transaction-specific data, or in other words the first item in the message stack.

The properties available to the `block`, `msg` and `tx` objects are listed in Table 2.8, Table 2.9 and Table 2.10 respectively. A built-in `gasleft()` function may also be called to return the amount of gas left.

Property	Type	Description
`block.number`	`uint256`	Current block's number
`block.timestamp`	`uint256`	Current block's unix timestamp
`block.gaslimit`	`uint256`	Current block's gas limit
`block.coinbase`	`address payable`	Address of the current block's beneficiary
`block.difficulty`	`uint256`	Current block's difficulty value

Table 2.8: List of `block` variable properties.

Property	Type	Description
`msg.sig`	`bytes4`	First four bytes of call data (function selector)
`msg.data`	`bytes`	Call data byte string
`msg.sender`	`address payable`	Message sender address
`msg.value`	`uint256`	Message Ether value

Table 2.9: List of `msg` variable properties.

Property	Type	Description
`tx.origin`	`address payable`	Sender of original transaction
`tx.gasprice`	`uint256`	Gas price of original transaction

Table 2.10: List of `tx` variable properties.

Error Handling

Solidity has two ways of handling errors or exceptional circumstances that require a state revert. The first method uses the low-level opcode `0xFD` (`REVERT`) to revert the current message and refund all remaining unspent gas to the caller. This opcode can be generated either by the `revert()` function or the `require(bool)` function that will only revert if the given boolean expression evaluates to `false`. Revert opcodes are also generated when, for example, funds are send to a non-`payable` function.

The second method of reverting state uses an invalid opcode, `0xFE`, and will revert execution without refunding gas. Invalid opcodes are generated by the built-in `assert(bool)` function whenever the boolean expression evaluates to `false`, and should be used to signal an invalid state or other broken invariant. Solidity will also generate these invalid opcodes for other exceptions, such as dividing by zero or passing an out of bounds index to an array.

Messages

Contracts in Solidity have several different ways of sending messages to one another depending on the needs of the developer. Table 2.11 lists all the built-in messaging functions available to an `address payable` object. The `send` and `transfer` functions should be used whenever the only requirement is that funds are transferred to the recipient using their fallback function, since these functions have a strict gas limit of 2300.

If it is necessary to provide more than 2300 gas or call data, then the more verbose `call` function may be used. The amount of gas and Ether value sent with the `call` message may be set by chaining the optional `gas(uint256)` and `value(uint256)` functions, for example:

```
result = recipient.call.gas(4000).value(10000)();
```

Omitting a data string as function payload will execute the fallback function in the receiving contract.

The other variants of the `call` function, `staticcall` and `delegatecall`, share the same syntax for controlling the gas and Ether value of the message call, but differ in the way the recipient contract may act on the blockchain. Recipients of a `staticcall` are not allowed to write any information to storage and are intended to be used only for pure functions with no side-effects. A `delegatecall` is used to allow the recipient's contract code to interact with the sender's storage and can naturally be very dangerous if the recipient address is somehow manipulated by an attacker.

One example use case of the delegatecall function is to handle the migration and release of new contract code in a production environment. The calling contract may contain the persistent data, while the recipient of the delegatecall is updated with new logic by authorised users [16].

All calls described so far will halt execution as it waits for a message to return, but any changes made thus far to its storage will be visible to other contracts within the system — should they make another call back to the original contract. Re-entrant calls like this are considered a major vulnerability and is the focus of Section 3.2.3 on page 33.

Function	Description
`send(uint)`	Send the specified amount of Ether to this address and return a boolean indicating success.
`transfer(uint)`	Send the specified amount of Ether to this address and revert if it fails.
`call(bytes)`	Call this address with the specified payload and return a boolean indicating success.
`staticcall(bytes)`	Call this address with the specified payload while allowing no changes to its state and return a boolean indicating success.
`delegatecall(bytes)`	Call this address with the specified payload while providing it with access to the current contract's state and return a boolean indicating success.

Table 2.11: List of Solidity messaging functions for payable addresses.

2.2.7 Token Standards

An Ethereum Request for Comment (ERC) can be used to specify a standard interface that developers can implement in their own contracts, the most pop-

ular of which is the ERC-20 fungible token standard. Table 2.12 contains all the functions that have to be implemented for a contract to comply with this standard [17]. This standard is commonly used to build initial coin offering (ICO) tokens and reduces the time and complexity of developing one's own tradable token. By relying on a common standard, developers further aid those auditing the ICO for vulnerabilities or others building external wallet software that track ERC-20 token balances using this standardised interface.

Signature	Returns	Description
`name()`	`string`	Name of the token (optional)
`symbol()`	`string`	Symbol of the token (optional)
`decimals()`	`uint8`	Number of decimals for user representation (optional)
`totalSupply()`	`uint256`	Number of total tokens in supply
`balanceOf(address)`	`uint256`	Number of tokens owned by the specified address
`transfer(address,uint256)`	`bool`	Transfers your tokens to the specified address. Emits a `Transfer` event
`transferFrom(address, address,uint256)`	`bool`	Transfers tokens from one address to another. Emits a `Transfer` event
`approve(address,uint256)`	`bool`	Allows the address to transfer on your behalf. Emits an `Approve` event
`allowance(address,address)`	`uint256`	Number of tokens that one account my withdraw from another

Table 2.12: List of ERC-20 token standard functions.

Alternatively, developers can implement the more recent ERC-721 standard [18], which implements a non-fungible token. This allows one to extend the token and add their own functionality, while still identifying every token in circulation with a unique ID.

2.3 Solidity Development and Testing

Having explained the EVM and Solidity language in the previous two sections, we now briefly present ways in which a smart contract developer can test their code before it is deployed on the main Ethereum network. The tooling

discussed in this section are either the most popular solutions or officially supported by the Ethereum development team.

2.3.1 Test Network Clients

Like most software systems, deploying and running a smart contract in its production environment (the main Ethereum network) can be prohibitively expensive and time consuming. Developers therefore need test environments running a simulated Ethereum blockchain that focus on speed, accuracy and debugging information.

The officially developed mining clients, for example:

- Geth (written in Go) [19],
- Aleth (written in C++) [20], and
- Py-EVM (written in Python) [21],

allow users to run local versions of the blockchain with the option of mimicking its consensus protocol during mining. Another popular community option is Ganache [22], which offers another local, fast implementation of the Ethereum network and a graphical user interface for exploring transactions. These clients all implement a standardised set of RPC methods that interact with the blockchain, but clients such as Ganache implement additional calls that also revert the network's state to a previous snapshot — making it ideal for testing.

There are also test networks supported by the community that more closely resemble the main network, except they provide users with a free supply of Ether [23].

2.3.2 Development Frameworks

One of the most popular development frameworks for developing smart contracts is the Truffle suite [7]. This fully fledged environment assists with, amongst other features, the following:

- Compilation and binary asset management for smart contracts;
- Scripting and management of connected public and private networks (including Ganache as its standard test option);
- Integration testing scripts with JavaScript;
- Deployment and migration management;
- Interactive console and other useful commands; and
- APIs to integrate external plugins or scripts.

2.3.3 Unit Testing

Truffle provides the ability to write unit tests in Solidity and offers a variety of helpful libraries to test authors. These tests are written in Solidity as regular contracts named with a `Test` prefix and contain functions that also start with `test`. Running the test suite with the `truffle test` command will compile the test suite and run it on the selected Ethereum client.

Although truffle will ensure the Ethereum client loads a clean environment each time (either by resetting the chain or reverting to a previous snapshot), one can also define before and after hooks using additional built-in functions. Additional Solidity libraries are also packaged with Truffle that offer more flexible, in-depth test assertions (under `truffle/Assert.sol`) and help with dynamically managing the addresses of deployed contracts at runtime (under `truffle/DeployedAddresses.sol`). The following is a very simple unit test example included in the Truffle suite documentation that demonstrates the assertion library and test hooks:

```
1  import "truffle/Assert.sol";
2
3  contract TestHooks {
4    uint someValue;
5
6    function beforeEach() {
7      someValue = 5;
8    }
9
10   function beforeEachAgain() {
11     someValue += 1;
12   }
13
14   function testSomeValueIsSix() {
15     uint expected = 6;
16
17     Assert.equal(someValue, expected, "someValue should have been 6");
18   }
19 }
```

2.3.4 Integration Testing

Integration tests can also be written as part of the Truffle suite using the Mocha testing framework [24]. These tests are written in JavaScript and allow developers more control over the blockchain environment for each test, since they can now use the popular web3 API [25] to deploy contracts, submit transactions, read blockchain information, etc.

Developers can make use of the `artifacts.require()` JavaScript function to load contract definitions from the current project and then use the `contract()` function to define each test case. Test cases defined using this function behave similarly to Mocha's `describe()` function, but with the added benefit of resetting the client's Ethereum environment and ensuring it is reverted to a clean snapshot before each test. The Chai assertion library [26] can also be used to further check that the system is running correctly.

Listing 3 shows another test case snippet from the Truffle documentation, but now showing how the MetaCoin contract can be deployed and checked for an Ether balance assertion.

```
const MetaCoin = artifacts.require("MetaCoin");

contract("MetaCoin", accounts => {
  it("should put 10000 MetaCoin in the first account", () =>
    MetaCoin.deployed()
      .then(instance => instance.getBalance.call(accounts[0]))
      .then(balance => {
        assert.equal(
          balance.valueOf(),
          10000,
          "10000 wasn't in the first account"
        );
      }));
});
```

Listing 3: Metacoin example using the Mocha and Chai libraries.

2.4 Fuzzing

Fuzzing is a method of automated testing that tests a program using large amounts of procedural input data. The fuzzer (or fuzz tester) can typically feed large amounts of input data to the system, analyse the output, and then potentially generate new input in such a way that a different program execution path is followed. If any interesting or vulnerable program states are reached (such as a crash or memory leak), then the fuzzer will be able to report the trace of inputs necessary to reach the state and recreate the problem.

Fuzzers can broadly be differentiated from one another by their method of input generation and the type of feedback returned by the program, as explained further in Sections 2.4.1 and 2.4.2 respectively.

2.4.1 Input Generation

Program inputs from fuzzers are generated typically using one of two ways, the first of which generates it using a known structure associated with its valid input either provided by the user or analysed by the fuzzer. This can take the form of, for example, a grammar-based [27] or model-based [28] approach. The main advantage to this approach is that the fuzzer could use this additional knowledge of the system's valid input to spend less time stressing parsing components and explore potentially more interesting, deeper program paths.

The other popular approach involves continuously mutating the initial seed input and generating large amounts of test data. Examples of the mutation algorithm include flipping random bits, using interesting values such as the minimum or maximum available integer, or the more complex genetic algorithms used by a tool such as AFL [29].

2.4.2 Execution Feedback

Another important form of distinction among fuzzers is the amount of information they have about the given system under test and the feedback that they gain during fuzzing. We can broadly define these fuzzers into the following three categories of increasing levels of knowledge about the system:

- Black-box fuzzers only see the program output with no other knowledge of the system;
- Grey-box fuzzers employ some form of instrumentation to gain coverage or execution trace feedback, e.g AFL;
- White-box fuzzers have full access to the program source code and could therefore perform static analysis to gain more information about the system, e.g. SAGE [30].

This feedback and additional knowledge of the system can be used to inform the input generation component of the fuzzer, thereby exploring more paths within the system.

In the case of fuzzing the EVM, we typically have at a bare minimum access to the full trace of each execution and, in some cases, publicly verified source code written in high-level languages like Solidity.

2.5 Symbolic Execution

Symbolic execution is an analysis technique that determines under what constraints on the input a certain program path would be executed. In this

section we will briefly explain the concept of symbolic execution and the underlying methods crucial to understanding the tools discussed in later chapters. Tool-specific discussion and an analysis of the different properties that can be checked using symbolic engines is in Chapter 4.

2.5.1 Example Program

A symbolic executioner begins by accepting certain specified program input parameters as symbolic values, instead of initialised or concrete values. The engine then starts executing the program, while keeping track of its state and the current path condition (PC). A path condition is a boolean formula that describes all the constraints on the initial symbolic input that leads to the current program state. Each of these explored paths can be combined into a symbolic execution tree, where the nodes represent possible program states and the edges represent additional constraints appended to the path condition to reach those new states [31].

As an example, consider the function in Listing 4. Initially when we start executing this function, our parameters (`x` and `y`) will be treated in the program state as symbolic values. The symbolic engine will proceed to step through each instruction and simulate the results, but fork at conditional jumps. Each jump's conditional expression is appended to the path condition of the next state and continued in this manner until execution terminates. Figure 2.1 shows the potential symbolic execution path for this example function with the variable values and path conditions at each state. The only infeasible path condition is that of the bottom left state (`X > Y && Y - X > 0`), therefore indicating that the assert statement is impossible to reach and the function is safe from this type of error.

```
1 function swap(int x, int y) {
2     if (x > y) {
3         x = x + y;
4         y = x - y;
5         x = x - y;
6         if (x - y > 0)
7             assert(false);
8     }
9 }
```

Listing 4: Example swapping function.

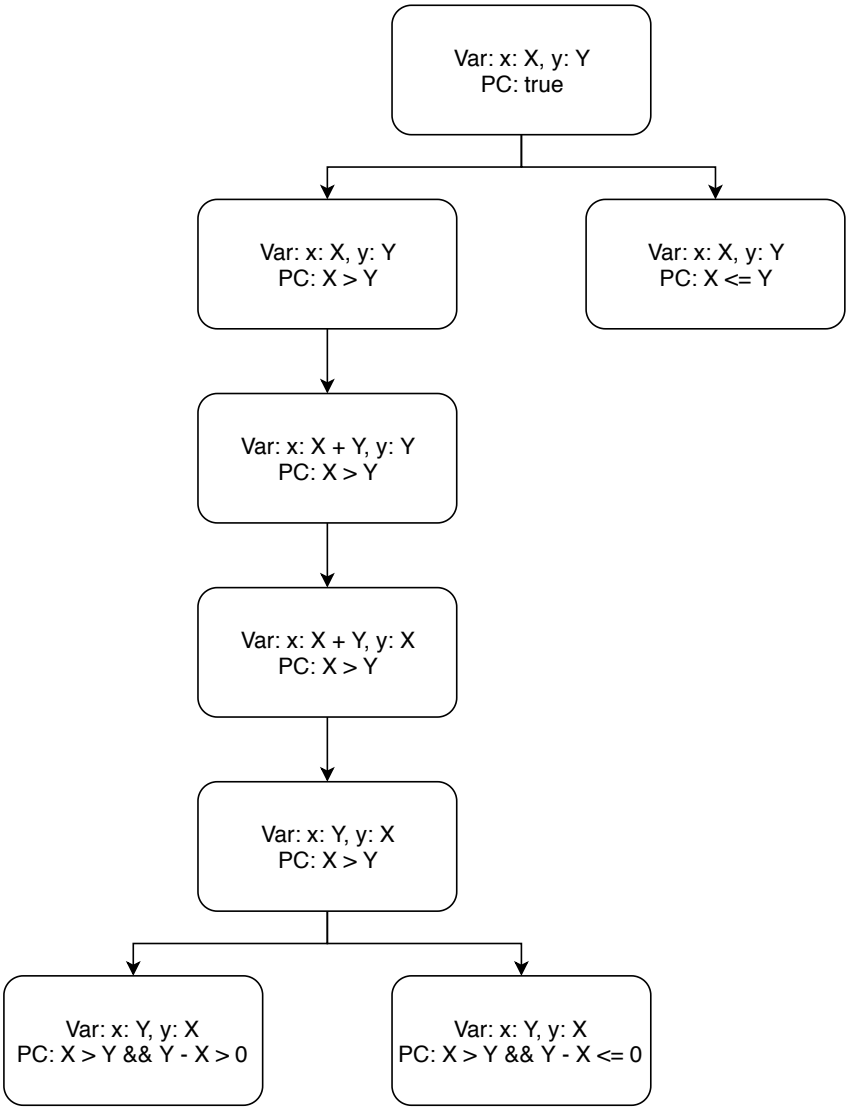

Figure 2.1: Symbolic execution tree of the example swapping function showing the symbolic variables stored in memory and current path condition at each state.

Chapter 3

Vulnerabilities

In this chapter we introduce and explain the most common vulnerabilities present in Ethereum smart contracts. We assume that the reader has a basic understanding of the EVM, as described in Chapter 2. While many of the examples are not specific to Solidity and purely rely on the EVM bytecode, we may opt to provide example source code written in Solidity where appropriate. Certain vulnerabilities and attacks related to the blockchain itself (e.g. replay attacks) fall outside the scope of this section and will not be considered unless it depends on the EVM bytecode or a Solidity contract.

3.1 Vulnerability Taxonomies

The first taxonomy presented in Table 3.1 was initially proposed by Atzei et al. in a survey published in 2017 [32] and adapted by Ardit Dika that same year [33]. It classifies vulnerabilities according to the level in the Ethereum programming stack that they are introduced at (Solidity, EVM or the blockchain itself) and a severity level as determined by their research and prior occurrences. Since the majority of the tools analysed in later chapters are able to only accept EVM bytecode as input, we felt that the focus had to be shifted towards the vulnerabilities that are not reliant on Solidity specific practices.

We therefore draw on this existing taxonomy, along with other surveys [34], and registries maintained by organisations or members within the industry, [35] and [36], to present a new taxonomy in Table 3.2. This taxonomy focuses on severe vulnerabilities that still exist at the Ethereum application layer and group similar ones into four categories. The vulnerabilities are primarily named after the ones given in the SWC Registry [35] to maintain a single common naming scheme.

Level	Vulnerability	Severity
Solidity	Gas costly patterns	Low-Medium
	Call to the unknown	High
	Gasless send	High
	Exception disorders	High
	Type casts	Medium
	Re-entrancy	High
	Unchecked math (integer over and underflow)	Low-Medium
	Exposed functions or secrets	Medium-High
	tx.origin usage	High
	blockhash usage	Medium-High
	DoS	High
	send instead of transfer	Low-Medium
	Style violation	Low
	Redundant fallback function	Low
EVM	Immutable bugs	High
	Ether lost in transfer	High
Blockchain	Unpredictable state	Medium
	Generating randomness	Medium-High
	Timestamp dependence	Low-High
	Lack of transactional privacy	Low-High
	Transaction-ordering dependence	Medium-High
	Untrustworthy data feeds (oracles)	High

Table 3.1: Taxonomy of vulnerabilities based on the level that they are introduced at and severity.

3.2 List of Vulnerabilities

This section lists each of the major vulnerabilities that were listed in our taxonomy in the previous section and describes the nature of the particular vulnerability, provides an example of a vulnerable smart contract, and demonstrates how to exploit it. Best practices or methods of protecting against a vulnerability will also be included where applicable.

Section 3.3 lists the vulnerabilities that were omitted from our taxonomy, but still included since some of them are checked by the analysis tools discussed in Chapter 4.

3.2.1 Integer Overflow

All arithmetic operations in the EVM, whether signed or unsigned, can overflow after reaching the 256-bit internal word size limit and wrap around to zero. Any use of the `ADD`, `SUB`, `MUL` or `EXP` opcodes that accepts a value that was

Category	Vulnerability
Access Control	Unauthorised Ether Withdrawal
	Unauthorised Self-destruct
	Unauthorised Delegatecall
	Authorisation Through Origin
Denial of Service	DoS with Failed Call
	DoS with Block Gas Limit
	DoS from Greedy State
Missing Checks	Write to Arbitrary Storage Locations
	Integer Overflow
	Re-entrancy
	Unchecked Call Return Value
Requirement and Invariant Violations	Transaction Order Dependence
	Weak Sources of Randomness
	Assert Violations
	Ether Invariants

Table 3.2: Taxonomy of vulnerabilities grouped according to categories.

either directly supplied by the user or tainted by another user-supplied value is therefore a potential vulnerability. There is currently no EVM instructions that reports if an overflow has occurred or not and thus this must be checked by the developer.

Prevention

The simplest way of preventing integer overflow is to analyse every potentially vulnerable arithmetic result and revert the current transaction if an overflow has occurred. A popular example of this check is the open-source `SafeMath` library developed by the OpenZeppelin organisation. Listing 5 includes the safety check for addition of two 256-bit unsigned integers, which calculates the addition result of the two parameters and reverts if the result is smaller than the first parameter (i.e. it has wrapped around). The rest of the library contains similar functions for subtraction and multiplication of 256-bit integers, as well as two functions that check for division and modulo of zero operations.

3.2.2 Unchecked Call Return Value

Message calls in the EVM return a boolean value to indicate the success of the call, with zero indicating failure. The built-in `call`, `send`, `delegatecall` and `staticcall` functions all make use of a `bool` return value to indicate success, while the `transfer` function instead reverts when a message call fails. It is

```
library SafeMath {
    function add(uint256 a, uint256 b) internal pure
       returns (uint256) {
          uint256 c = a + b;
          require(c >= a, "SafeMath: addition overflow");
          return c;
    }
    ...
}
```

Listing 5: Abridged `SafeMath` library.

considered good practice to always check the result of these calls and handle the case where a call might fail for whatever reason.

A failed call could, for example, indicate that the recipient contract ran out of gas during execution. The contract author would have to account for this potential scenario and either gracefully revert the current transaction or ensure that the same contract is not called again.

Prevention

Always consider the case where a call using one of the functions mentioned above might fail and then handle it appropriately. In general this means using the call status return variable in an `if-else` conditional and either reverting the entire transaction or manually reverting the contract state to the point just before the call was made.

The Solidity snippet in Listing 6 shows an example of a transfer of funds to the recipient address with a conditional clause that reverts the internal balance should the message fail. If the contract requirements only dictate that its internal state is reverted back to its state prior to the transaction, then the `transfer` function is a more concise alternative.

```
amount = balances[recipient]
balances[recipient] = 0;

if (!recipient.send(amount)) {
    balances[recipient] = amount;

    // continue with rest of the error handling code
}
```

Listing 6: Call return value handling example.

3.2.3 Re-entrancy

Message calls in the EVM switches control flow to the recipient contract while still maintaining the sender's substate. Any state changes that were made during the sender's initial execution before the message call are therefore visible to the recipient contract and can be exploited by a subsequent re-entrant call. Given enough gas, the recipient can potentially invoke one or more functions in the original sender contract multiple times to eventually drain a significant amount of its Ether balance — as was the case in the now infamous DAO hack.

Listing 7 contains a simple re-entrancy example from the SWC Registry maintained by ConsenSys [35]. The following steps are necessary to exploit this contract and generally form the basis of most attacks of this kind:

1. The attacker deposits some amount of Ether or otherwise performs an action that will later entitle them to an authorised Ether withdrawal. In this case they call the `donate` function.

2. The attacker calls some form of withdrawal function (`withdraw` here) that initiates a transfer to their account with enough gas to perform other tasks.

3. The attacker receives this amount, but may now execute their own code while the vulnerable contract is still in an intermediate substate.

4. The attacker contract calls the same function and is once again allowed to withdraw the same amount, since the state variable representing the balance will only be updated after the first call has finished.

5. The attacker is free to repeat this process until the other contract is virtually drained of all its Ether and then return control flow to finish the transaction.

An alternate variation of this exploit involves the attacker making a re-entrant call while the original contract remains in an invalid substate with the aim of modifying storage in such a way that they may setup another attack in a subsequent transaction.

There is one crucial mistake made in this example contract and that is that its state is only updated after the call has finished, thus leaving it in an intermediate state that violates one of its invariants (the sum of all the user balances equals the contract's own Ether balance). A developer could avoid re-entrancy in this simple example by limiting the amount of gas transferred to the withdrawal recipient, however, this negates one of the strengths of the Ethereum platform and might not adhere to the requirements of a large system of connected contracts.

We therefore define two potential scenarios where a re-entrancy attack may have occurred during a transaction:

```
pragma solidity 0.4.24;

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

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

  function withdraw(uint amount) public{
    if (credit[msg.sender]>= amount) {
      require(msg.sender.call.value(amount)());
      credit[msg.sender]-=amount;
    }
  }

  function queryCredit(address to) view public returns(uint){
    return credit[to];
  }
}
```

Listing 7: SWC Registry re-entrancy example contract.

- Storage is modified during a re-entrant call; or
- A contract's Ether balance is less than expected after the successful transfer of funds.

Prevention

There are several ways to protect a contract against re-entrancy. The best practice in terms of gas cost and functionality is to make all intended changes to the contract's state before initiating the first message call and revert after if it fails, otherwise known as the "checks-effects-interaction" pattern [37]. The `withdraw(uint)` function in the example above may be rewritten as follows to maintain a valid state during the call and prevent the re-entrant call from withdrawing more Ether than allowed:

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

This still allows the recipient contract to perform other tasks, but the balance in the original contract will remain consistent even if the `withdraw` function is called again.

Another solution simply involves limiting the gas stipend sent along with every outgoing message call. A message sent using Solidity's `call` function will forward all available gas to the recipient, but this may be overridden using the `.gas(amount)` method, for example:

```
recipient.call.gas(70000).value(amount)()
```

Message calls in the EVM have an inherent base fee of 21000 gas associated with storage and signature operations, so limiting the stipend to this amount will prevent an attacker from making another call. If the only goal is to transfer funds, then using `send` or `transfer` is considered best practice, since these two functions only provide a 2300 gas stipend (sufficient for basic logging tasks).

The last option maintains the ability for other contracts to perform their own calls, but implements a mutex that prevents any re-entrant calls from interfering with the contract while it is in a substate [37]. Listing 8 includes an abridged version of the `Mutex` contract example included within the official Solidity documentation. Any function implemented with this modifier will be locked using a state variable and may only be unlocked once the entire function call has finished. This strategy raises the gas cost of execution due to the extra storage operations, but may prove useful if, for example, a function requires a sequence of transfers with more complex state changes.

```
1  contract Mutex {
2      bool locked;
3
4      modifier noReentrancy() {
5          require(!locked);
6          locked = true;
7          _;
8          locked = false;
9      }
10 }
```

Listing 8: Mutex modifier example.

3.2.4 Assert Violations

Although the EVM only specifies one type of revert instruction, namely (`0xFD REVERT`), Solidity allows developers to also cause a state reversal based on an

assertion failure by using an invalid opcode (`0xFE`). The `REVERT` opcode also causes the sender to be refunded all the remaining gas, where an invalid opcode does not.

A popular use case for the `assert` function is therefore to check for violations of some internal invariant or to detect bugs, while `require` is the preferred function to deal with any known exceptions (e.g. invalid user input or failed calls). Program execution logs can be monitored during testing and any instance of the `0xFE` opcode will indicate the presence of a bug and allow the developer to address the issue.

Prevention

Assert statements are generally used during the development and testing stages of software engineering to detect bugs or other unintended program states. If there are still uses of the `assert` function present before a public release, then it would be best to log the failure by emitting a Solidity event and using a `revert` statement to roll back the program state and refund the user their remaining gas.

3.2.5 DoS with Failed Call

In Section 3.2.2 we discussed the issue of ignoring a call return value — typically in the context of a single message call. We now consider the scenario where contract operations could be stalled or permanently halted by a malicious recipient that always fails a transfer on purpose or due to the low gas stipend of some of the built-in Solidity functions. The simplest example of such a contract is one that contains a Solidity default function that always reverts when called, as in the following example:

```
contract Fail {
    function () {
        revert();
    }
}
```

Listing 9 contains a modified example contract from the Ethereum wiki implementing a simple auction system. In this example each bidder must call the `bid` function with an Ether value greater than or equal to the previous highest bidder. If a lower Ether value is sent, the transaction immediately reverts due to the `require` function, otherwise it accepts the payment and refunds the previous highest bidder. Due to the use of the `transfer` function during the refund process, the entire transaction will revert if the previous bidder's address contains a contract that intentionally fails or consumes more than the 2300 implicit `transfer` gas stipend within its default function.

```
pragma solidity 0.5.10;

contract Auction {
    address payable highestBidder;
    uint highestBid;

    function bid() external payable {
        require(msg.value >= highestBid);

        if (highestBidder != address(0)) {
            highestBidder.transfer(highestBid);
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }
}
```

Listing 9: Auction contract example.

Prevention

The first step in protecting against a denial of service of this nature is to analyse the sections of the contract that sends messages to beneficiaries. Every call must be assumed to be able to fail and return `false` when using the `send` or `call` functions or to revert when using `transfer`.

Often the reality is that the contract in question can not move forward even if a message failure is considered. In our contract above, accepting a new bid requires that the previous highest bidder be refunded their deposit, so the two options here are to either forfeit that amount or implement a way for bidders to collect their funds at a later stage.

This practice of shifting the responsibility of withdrawal to the beneficiary is commonly referred to as a "Pull over Push" strategy. Instead of immediately paying a beneficiary or a set of beneficiaries, the contract must implement a balance for each account and allow them to request a withdrawal in a separate transaction. Listing 10 contains the previous `Auction` contract shown above, but now with an internal mapping for each bidder's refund amount and a function allowing them to withdraw that amount. The contract may now continue accepting new bids even if one of the bidders intentionally fail their withdrawal. Such a rewrite of the system does, however, come with a few drawbacks:

1. The contract now requires substantially more storage on the blockchain due to the `refunds` map;

```
pragma solidity 0.5.10;

contract Auction {
    address payable highestBidder;
    uint highestBid;
    mapping(address => uint) refunds;

    function bid() external payable {
        require(msg.value >= highestBid);

        if (highestBidder != address(0)) {
            refunds[highestBidder] += highestBid;
        }

       highestBidder = msg.sender;
       highestBid = msg.value;
    }

    function withdrawRefund() external {
        uint refund = refunds[msg.sender];
        refunds[msg.sender] = 0;
        msg.sender.transfer(refund);
    }
}
```

Listing 10: Auction contract example with withdraw functionality.

2. The overall gas cost is increased due to the additional bookkeeping and withdrawal operations;

3. The number of transactions on the blockchain will increase, with the cost shifted to the users requesting the withdrawal; and

4. The overall user experience is diminished by requiring them to initiate the withdrawal, assuming that each bidder withdraws promptly (if at all).

3.2.6 DoS with Block Gas Limit

Ethereum has two systems in place to ensure programs always halt:

- A stack size limit of 1024 items; and
- A variable gas limit for the entire block of transactions.

The block gas limit determines the maximum number of transactions that are allowed per block and is voted on by the mining community. As of the time of writing, this limit hovers around 8 million gas units.

These limits therefore allow for a contract to potentially enter a state where it is impossible to call a specific function, due to its computational requirements. Generally this can happen whenever a function loops over an array whose length is controlled by the contract users, as shown in the `totalBalance` function in Listing 11. Assuming no other way to remove an address from the list of creditors, this array can grow indefinitely as time goes on and no longer be callable within the block gas limit. Naturally the risk increases as the complexity of the loop increases, whereas our example remains relatively cheap in terms of gas usage. Another potential pitfall is the use of the `call` function

```
contract Bank {
    address[] creditors;
    mapping(address => uint256) balances;

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

    function totalBalance() public returns (uint256) {
        uint256 total = 0;

        for (uint256 i = 0; i < creditors.length; i++) {
            total += balances[creditors[i]];
        }

        return total;
    }

    // Other functions
}
```

Listing 11: Mock banking contract with potentially infinite loop.

in Solidity that by default forwards all available gas to the recipient (capped to 63/64 of the remaining gas). An attacker may then load a contract that depletes all of its available gas during calls to its default function, thus leaving very little gas in the original calling contract after control flow is returned. If too many operations are performed afterwards, then it will cause the contract to revert due to insufficient gas.

Prevention

The primary approach to prevent an attack related to gas limits is to ensure that all functions have a deterministic upper bound in execution cost. This might mean that certain functions that involve looping over a list of values should either be able to be split into different blocks over multiple transactions or if possible stored as computed values in storage. A better solution might be to avoid loops altogether by adopting a strategy such as the "Pull over Push" one mentioned earlier.

Shifting operations such as a withdrawal to the user can also be a means around the insufficient gas issues, since only the user themselves will be affected if they load a malicious contract that is not capable of an error-free interaction. In all other cases, it is always good practice to consider the maximum possible gas cost of an external call and ensure that there will be enough left over to finish the transaction.

3.2.7 DoS from Greedy State

A contract is defined to be in a greedy state if it can receive Ether, but no longer transfer any of it to other accounts [38]. Funds can either be transferred to the contract using payable functions or forcibly deposited using self-destructs and mining fee transactions. Any funds received after entering this state are therefore considered frozen and permanently lost. If a contract does not contain any outbound call or self-destruct instructions, then it may be considered greedy by default, since there are no other methods to retrieve funds.

Contracts suffering from the other denial of service vulnerabilities described in Section 3.2.5 and Section 3.2.6 may further be classified to fall under this category if all withdrawal functions are permanently affected.

Prevention

The first step to protect against this vulnerability is to ensure that withdrawal functions do not suffer from any of the other DoS vulnerabilities such as the one produced by intentionally failed calls. One might also choose to include a self-destruct function to recover funds in the event that a contract is compromised, however, developers need to ensure that only authorised users can initiate this process (as discussed further in Section 3.2.9).

3.2.8 Unauthorised Ether Withdrawal

An unauthorised Ether withdrawal occurs when Ether funds are transferred to a recipient that can be entirely arbitrary or outside a set of trusted addresses. This may also be referred to as an Ether leak or a prodigal contract, although either terminology describes the same process by which an attacker may have

stolen funds using a message call. For the purpose of this vulnerability, we do not distinguish between calls that execute code in the recipient or calls that have empty data strings. As long as Ether is transferred successfully to an untrusted party via a call, it may be classified as a leak.

The precise definition of an untrusted contract typically varies from tool to tool. In general, if the recipient address is manipulable by the caller and able to evaluate to any arbitrary address, then it may be a cause for concern. The MAIAN tool [38], for example, reports this vulnerability when funds are transferred to an arbitrary address that:

- Does not own the contract under test;
- Has never deposited Ether to the contract under test; and
- Has "provided no data that is difficult to fabricate by an arbitrary observer".

Reporting on this vulnerability may be further improved by taking into account the severity of the Ether withdrawal by measuring whether or not the amount gained by the attacker is more than they deposited and measuring the percentage of the contract's total funds that can be withdrawn by the attacker.

Prevention

This vulnerability generally occurs as a result of another vulnerability that opens the contract up to unauthorised access (e.g. Section 3.2.14) or due to a lack of any authorisation controls. It is always vital to keep in mind that any contract created during the execution of another contract's code is visible to the public and callable by any account.

3.2.9 Unauthorised Self-destruct

The `selfdestruct` function in Solidity calls the `SELFDESTRUCT` opcode, which transfers all remaining funds to the specified address and removes the contract code and all its storage data from the blockchain. Any funds transferred to this self-destructed address will be frozen and remain permanently irretrievable, since the contract can no longer be modified to run new code.

We consider the criteria for an unauthorised self-destruct to be the same as that of an unauthorised withdrawal from Section 3.2.8, meaning that this function call must be preceded by sufficient checks for authorisation. The permanent loss of the contract is severe enough to warrant a report of this vulnerability, regardless of the recipient or amount of funds transferred.

Prevention

Functions that contain a self-destruct generally require some trusted address to initiate the call, otherwise they raise an exception. A very simple method of protecting against this is to store the message sender of the contract creation transaction as an `owner` variable and ensure that all calls to `selfdestruct` are preceded by a `require(msg.sender == owner)` statement. This pattern is provided by OpenZeppelin's `Ownable` base class that allows extended contracts to use a modifier with functions to check if the owner made the call or transfer ownership to a new address.

3.2.10 Unauthorised Delegatecall

The EVM provides a `delegatecall` opcode that allows one contract to execute another contract's code, while providing access to its storage. If the recipient address of the delegated call instruction can be manipulated by the attacker, then they would be able to load a contract that overwrites potentially critical storage data in the vulnerable contract. Any contract that therefore delegates a call to an arbitrary address is therefore considered vulnerable.

3.2.11 Transaction Order Dependence

Contracts are transaction order dependent if the outcome of a set of two or more of their transactions depend on their order within their block. In other words, the execution order may cause a race condition depending on the storage variables that are written or amounts of Ether transferred. This is an important concern for many types of contracts, because when a user submits their transaction to the network, they can not guarantee which block it will be accepted in or in what order it will be executed.

Listing 12 shows another version of the auction contract that we have been using, but with an added `stop` function that the owner can use to stop the auction. To demonstrate how a race condition can be exploited, assume we have two users (A and B) that are interested in bidding on the auction and our auction contract owner. The following tables show two different scenarios where the same transactions are ordered differently across two blocks:

Block 1
A bids 100
B bids 110
Owner stops

Block 2
A bids 100
Owner stops
B bids 110

Both scenarios have completely different outcomes and end up with two different values written to the `highestBidder`, despite user B clearly submitting the highest bid both times.

```
pragma solidity ^0.5.10;

contract Auction {
    address payable highestBidder;
    address owner;
    uint highestBid;
    bool stopped;

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

    function bid() external payable {
        require(!stopped);
        require(msg.value >= highestBid);

        if (highestBidder != address(0)) {
            highestBidder.transfer(highestBid);
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }

    function stop() external {
        require(msg.sender == owner);
        stopped = true;
    }
}
```

Listing 12: Auction example dependent on transaction order.

Prevention

Transaction order dependence can be prevented by implementing mutual exclusion locks, similar to the common concurrency problem of protecting against race conditions. Developers can, for example, store the current block number when entering a critical section and check afterwards in subsequent transactions to make sure that the block number has increased.

3.2.12 Authorisation Through Origin

The built-in Solidity value `tx.origin` returns the sender of the current transaction. Authorising functions using this value is therefore considered dangerous, because the value will remain the same across messages. If a transaction is originally made using the authorised `tx.origin` account, then any re-entrant calls made by attacking contracts will also read this same value and therefore potentially be able to initialise unauthorised operations.

Prevention

Any critical operations authorised for a specific address should instead use the `msg.sender` value, so that the value will more accurately reflect the sender of the current message instead of the original transaction.

3.2.13 Weak Sources of Randomness

Contracts attempting to implement random behaviour often end up making use of predictable blockchain or miner attributes [35]. There are several compounding factors that make using variables such as the current timestamp (`block.timestamp`) or block number (`block.number`) vulnerable to attack:

1. All contract data are publicly visible and can be studied by an attacker to determine the exact algorithm used to achieve pseudo-randomness; and

2. Colluding miners can choose what transactions to include in their blocks, the order of transactions, and even the exact timestamp up to a small margin before other miners submit their solution.

To demonstrate how attackers may choose to exploit this behaviour, consider the following public lottery function where the first user to submit at the correct time wins:

```
function play() public payable {
    require(msg.value > 1 ether);

    if ((block.timestamp * block.number) % 42 == 0) {
        msg.sender.transfer(address(this).balance);
    }
}
```

An attacker can now load a proxy account with a function that simply waits until the correct time to submit to the `lottery` contract and keep calling it until it is successful:

```
function attack() public {
    if ((block.timestamp * block.number) % 42 == 0) {
        lottery.play();
    }
}
```

Prevention

Contract authors always need to be careful when using these environmental variables and consider their true intention. If random behaviour is required, then contracts can make use of on-chain random number generators that rely on other users for generation (for example RANDAO [39]) or off-chain oracles that generate the numbers externally.

3.2.14 Write to Arbitrary Storage Locations

All persistent storage values in the EVM share the same segment, including values that are declared as part of a dynamic array in Solidity. Since these dynamic arrays can have a length of up to $2^{256} - 1$, an array with a sufficiently large length could therefore overwrite other variables within the storage space by wrapping around at the maximum unsigned integer size of $2^{256} - 1$. These collisions may go unnoticed and will not cause any exceptions during execution.

Listing 13 contains a contract that implement a dynamic array that can be manipulated (for the purpose of the example) by any user. During contract initialisation, the `owner` address variable will be set to the account that created the contract. This owner variable is written nowhere else in the contract and will be the only address that is trusted to successfully call the `destroy` function. The goal of this example exploit is to overwrite the owner variable using a sufficiently large dynamic array that collides with our other variables in storage. After initialisation and before any transactions are loaded, the variables are arranged in storage as follows due to the sequential order that they are declared in:

Offset	Variable	256-bit word
0	`owner`	`0xBf4eD7b27F1d666546E30D74d50d173d20bca754`
1	`a.length`	0

Any values at index i for an array declared sequentially at position n, will be stored at position

$$\text{keccak}(n) + i$$

using the Keccak-256 hash function. If we call `push(0)` in our example, then the value at the offset described above will be set to zero and the length of the array will be updated as follows:

```
contract Array {
    address payable owner;
    uint[] a;

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

    function pop() public {
        a.length--;
    }

    function push(uint val) {
        a.push(val);
    }

    function set(uint i, uint val) public {
        require(i < a.length);
        a[i] = val;
    }

    function destroy() public {
        require(msg.sender == owner);
        selfdestruct(owner);
    }
}
```

Listing 13: Example contract allowing arbitrary storage access.

Offset	Variable	256-bit word
0	`owner`	`0xBf4eD7b27F1d666546E30D74d50d173d20bca754`
1	`a.length`	1
...		
`keccak(1)`	`a[0]`	0

Our aim is to overwrite the variable at position 0, meaning we must choose a large enough index i for our array, such that

$$\text{keccak}(1) + i = 2^{256}.$$

One solution is to increase the size of the array during normal operations, but this may prove highly unfeasible due to the gas cost associated with that many transactions. The other option is to decrease the size of the array until the

length value wraps around at 0 and back to the maximum size of $2^{256} - 1$. If we call our `pop()` function twice, then the storage space will be ready to be exploited:

Offset	Variable	256-bit word
0	`owner`	`0xBf4eD7b27F1d666546E30D74d50d173d20bca754`
1	`a.length`	$2^{256} - 1$

All that is left is to call `set(i, val)` with $2^{256} -$ keccak(1) as the index value and the attacker's address as the unsigned integer value. Once successful, the `owner` variable will be overwritten and the attacker will gain access to the `destroy` function and be able to self-destruct the contract. All funds will subsequently be drained, despite trying to restrict this function only to a single owner address.

Prevention

Array lengths will in practice never reach the length required to collide with other variables, because of gas requirements (the number of transactions required will be orders of magnitude more than the number of Ethereum transactions to date).

Contract authors therefore need to make sure that contracts do not allow the user to decrement the array length below zero (e.g. `require(a.length > 0)`) or make use of the built-in `pop` function for arrays introduced in Solidity 0.5.0. The `a.pop()` function will delete the element from storage, decrement the length and trigger an invalid opcode exception if the array is already empty.

3.2.15 Ether Invariants

Developer assertions or other invariants that must hold during execution need to be carefully considered whenever evaluating a contract's Ether balance, since an account will receive Ether without triggering any code by way of another contract's self-destruction or a miner's coinbase transaction. Total user deposits therefore may have to be tracked using a separate variable if any invariants check assume that it will be the same as the contract's Ether balance.

Prevention

Invariants based on the contract's balance (`this.balance`) should always assume that it may be increased without explicitly sending a transaction to the contract in question. If the invariant is only meant to consider funds deposited by user accounts, then those amounts should be tracked separately during the call to each `payable` function.

3.3 Other Vulnerabilities and Bad Practices

This section briefly describes the vulnerabilities that were considered, but ultimately deemed outside the scope of this investigation. They are generally either too easily caught by static analysis tools, too low impact or too subjective and dependent on contract requirements to accurately evaluate as a vulnerability.

3.3.1 Solidity Compiler Version Issues

New Solidity compiler versions are frequently released to introduce bug fixes and performance enhancements, therefore making it best practice to target the latest stable release. It is also generally advised to not use a floating `pragma` statement at the top of a Solidity program and instead enforce a single compiler version, thus protecting against any subtle changes that might be introduced by new minor versions.

Since the tools discussed in the next chapter analyse the compiled bytecode directly, we can safely ignore these warnings and focus on potential vulnerabilities contained within the final code that is pushed to the blockchain.

3.3.2 Deprecated Functions

New Solidity compiler versions often deprecate and phase out functions or systems that cause vulnerabilities in the compiled bytecode. The compiler will either warn against the use of these functions or abort compilation in the case of a severe backwards compatibility issue.

In the end, our analysis tools discussed ultimately only consider the bytecode output and warn if it found a vulnerability. Source mappings are included to highlight the part of the Solidity code that may have generated the vulnerability, which the developer can then use to reconsider the compiler warnings and potentially use an updated function to avoid the vulnerability.

3.3.3 Unused or Uninitialised Variables

These are typically considered low impact vulnerabilities that are relatively easy to check. Uninitialised variables have well defined default values, meaning that the developer may intend to leave them uninitialised. If a vulnerability does arise because of these default values, then it will typically be caught further down the line when a more significant event occurs (e.g. unauthorised Ether withdrawal).

Similarly, an unused variable will only serve to clutter the codebase and potentially end up costing the user more gas if stored on the blockchain. It is

important to consider the cost to the user and optimise as much as possible, but it falls outside the scope of vulnerabilities that we intend to consider.

3.3.4 Use of Inline Assembly

While it is possible to write an inline assembly block in Solidity, it is generally warned against due to the fact that it bypasses many built-in Solidity features [6]. Using inline assembly can, however, allow experienced developers to optimise their code in terms of its gas requirements or perform operations that are not part of Solidity [40].

3.3.5 Timestamp Dependency

The use of the block timestamp is often highlighted separately from other sources of weak randomness, because the precise value of the timestamp is submitted by the miner and not necessarily the same as the exact time that the transaction was processed at. It is therefore recommended that developers not rely on the EVM timestamps and either use block numbers to estimate the passage of time or external oracles to provide a more precise measurement of time.

3.3.6 Signature Replay Attacks

One use case of Solidity and the EVM is to build smart contracts that rely on signature verification to perform certain tasks and authorise access. Developers could, for example, write a proxy wallet contract that transfers Ether to anyone presenting a valid signature communicated offline by the owner. This would function much like a cheque and a traditional bank account.

Ethereum allows accounts to verify that a message was signed by its associated private key, but it is vital that extra information about the nonce and proxy address is encoded in messages to avoid replay attacks [41].

Chapter 4

Tools

The previous chapter gave an overview of all the major vulnerabilities that exist in Ethereum smart contracts and examples of how to prevent some of them. We now focus on the testing and verification tools that are at our disposal to automatically find these vulnerabilities before contracts are deployed to a live blockchain.

Our first section in this chapter gives a brief overview of all the tools discussed in the rest of the thesis and outlines their features. In Section 4.2 we discuss each investigated tool in more depth and focus on the types of vulnerabilities that they are able to automatically detect. Our selection criteria for the tools that form part of this discussion are those that:

- Are in active development;
- Have a simple user-interface or continuous integration option;
- Are offered as an extensible open-source application or provide a developer API; and
- Produce detailed reports about the vulnerabilities encountered.

Unit and integration testing frameworks were deemed to fall outside the scope of this chapter, since they require the user to provide concrete program inputs. We do, however, give an overview of unit and integration testing of Solidity programs in Section 2.3 of our background chapter on page 22.

4.1 Tool Taxonomies

We present two taxonomies that classify our selected tools based on a set of significant features and the vulnerabilities that they are able to automatically detect. Table 4.1 lists the feature classification and Table 4.2 the vulnerability classification.

The next section will discuss each tool in more detail, but it is worth noting that MythX supports several features partially, since it is composed of more than one tool — one of which is Mythril. It can therefore only generate test inputs when a vulnerability was detected using its fuzzer or symbolic analysis component and not when using static methods.

Feature	Securify	Slither	MythX	Echidna	Manticore	Mythril
Open-source	Yes	Yes	Partially	Yes	Yes	Yes
Official API Support	No	Yes	Yes	Yes	Yes	Yes
Custom Property DSL	No	No	No	No	No	No
Test Input Generation	No	No	Partially	Yes	Yes	Yes
Coverage Analysis	No	No	No	Yes	Yes	Yes
Truffle Integration	Yes	Yes	Yes	Yes	Yes	Yes

Table 4.1: General feature classification of all the tools considered.

Vulnerability	Securify	Slither	MythX	Manticore	Mythril
Unauthorised Ether Withdrawal	Yes	Yes	Yes	Yes	Yes
Unauthorised Self-destruct	Yes	Yes	Yes	Yes	Yes
Unauthorised Delegatecall	Yes	Yes	Pro	Yes	Yes
Authorisation Through Origin	Yes	Yes	Pro	Yes	
DoS with Failed Call		Yes	Pro		Yes
DoS with Block Gas Limit			Pro		
DoS from Greedy State	Yes	Yes			
Write to Arbitrary Storage Locations			Pro		
Integer Overflow			Yes	Yes	Yes
Re-entrancy	Yes	Yes	Pro	Yes	Yes
Unchecked Call Return Value	Yes	Yes	Yes	Yes	Yes
Transaction Order Dependence	Yes		Pro	Yes	
Weak Sources of Randomness			Pro	Yes	Yes
Assert Violations			Yes	Yes	Yes
Ether Invariants				Yes	

Table 4.2: Vulnerability classification of all the tools considered. The MythX vulnerabilities marked “Pro” are only available in the paid subscription tier.

4.2 List of Tools

In this section we give a brief overview of all of the selected tools. Each tooling subsection will include a brief technical discussion based on their available literature and a full list of their detectable vulnerabilities. We will also mentioned the viability of adding extensions or implementing custom properties — where applicable.

4.2.1 Securify

The Securify security analysis tool checks a given Ethereum smart contract for the violation of a variety of properties that signal a vulnerability in the code. This tool was published in 2018 [42] and is available online via a web interface [43] hosted by the Chainsecurity organization [44]. Source code for the tool is also available at its Github page [45], which includes a Java command-line application.

Its web tool accepts a contract using a text, zip file, or git repository uploader and produces a line-by-line analysis report that either warns about a potential vulnerability or indicates a sure violation according to its pre-defined properties. The report provided by this web tool does not present a sequence of transactions that exploit the property violations.

Implementation Overview

The Securify analysis system requires two sets of input: EVM bytecode and security patterns written in its domain-specific language (DSL). This EVM bytecode is first decompiled into a static single assignment (SSA) form that captures a stackless intermediate representation (IR) of the program. Securify then infers semantic facts about the program by analysing its data and control-flow graphs and declares them in stratified Datalog. All available security patterns (provided by the user or built-in) are then checked for logical violations and reported to the user on a line-by-line basis.

Security Properties

Securify's property names are mapped to the vulnerability names used in the previous chapter in Table 4.3, while Table 4.4 contains the full list of properties checked by Securify's built-in security patterns, along with their description of each. Each pattern is written internally in both a compliance form (implying that it is satisfiable at all times) and violation form (implying that it can be found unsatisfied in some state). Securify vulnerability reports thus fall into three categories:

Violations The vulnerability's violation pattern was satisfied and the contract is deemed unsafe.

Warnings The violation pattern could not be satisfied, but neither could the compliance pattern.

Safe All compliance patterns hold while none of the violations are satisfiable.

Our Name	Securify Name
Unauthorised Ether Withdrawal	Unrestricted Ether flow
Unauthorised Self-destruct	Unrestricted Selfdestruct
Unauthorised Delegatecall	Potentially Dangerous Delegatecall
Authorisation Through Origin	Use Of Origin
DoS from Greedy State	Locked Ether
Re-entrancy	Reentrant method call
Unchecked Call Return Value	Unhandled Exception
Transaction Order Dependence	Transaction Order Affects Ether Amount Transaction Order Affects Ether Receiver

Table 4.3: Mapping between our vulnerability names and the closest Securify property names.

Property Name	Description
Reentrant method call	Method calls that are followed by state changes may be reentrant.
Gas-dependent Reentrancy	Calls into external contracts that receive all remaining gas and are followed by state changes may be reentrant.
Reentrancy with constant gas	Ether transfers (such as send and transfer) that are followed by state changes may be reentrant.
Missing Input Validation	Method arguments must be sanitized before they are used in computations.
Unrestricted ether flow	The execution of ether flows should be restricted to an authorized set of users.
Unrestricted write to storage	Contract fields that can be modified by any user must be inspected.
Unrestricted Selfdestruct	The execution of selfdestruct statements must be restricted to an authorized set of users.
Unsafe Call to Untrusted Contract	The target of a call instruction can be manipulated by an attacker.
Unsafe Dependence On Block Gas	Security-sensitive operations must not depend on gas-related information.
Potentially Dangerous Delegatecall	The inputs provided to delegatecall must be sanitized to avoid risk.
Repeated Calls to untrusted code	Repeated Calls to untrusted code might return inconsistent results.
Repeated call to an untrusted contract	Repeated call to an untrusted contract may result in different values
Unhandled Exception	The return value of statements that may return error values must be explicitly checked.
Division Before Multiplication	The use of division before multiplication may result in incorrect final results due to integer rounding.
Division influences Transfer Amount	The use of division to calculate the amount of transferred ether may be incorrect due to integer rounding.
Locked Ether	Contracts that may receive ether must also allow users to extract the deposited ether from the contract.
Use Of Origin	The origin statement must not be used for authorization.
Transaction Order Affects Ether Amount	The amount of ether transferred must not be influenced by other transactions.
Transaction Order Affects Ether Receiver	The receiver of ether transfers must not be influenced by other transactions.
Transaction Order Affects Execution of Ether Transfer	Ether transfers whose execution can be manipulated by other transactions must be inspected for unintended behavior.

Table 4.4: List of built-in security properties checked by Securify.

4.2.2 Slither

Slither is a static analysis framework [46] developed by the Trail of Bits cybersecurity consultation firm [47]. This static analyser automatically detects and alerts the user to a variety of vulnerabilities, bad coding practices, and

optimisation options present in the given code. It is offered as a command-line application, continuous integration service (as part of the Github app [48]), and as a Python API where developers can add their own vulnerability detector modules.

Implementation Overview

The first stage of the Slither analysis process compiles the given smart contract using the Solidity compiler and retrieves its abstract syntax tree (AST). Afterwards the contract inheritance structure, control flow graph (CFG) and Solidity expressions are recovered and the contract code is transformed into an SSA IR of the contract called SlithIR. Additional analysis is performed to determine, for example, the variables written and read or potential data dependency between transactions, before lastly running each available detector module. These detector modules can fully explore all the static analysis components gathered up to this point and output meaningful information or warning to the user in the event that a vulnerability is detected.

Security Properties

Slither comes with many built-in detectors aimed at finding vulnerabilities and optimisation opportunities or assisting with code review and understanding. These modules can be freely enabled or disabled when using the command-line interface or Python API. Developers may also write their own detectors for use with the API to detect custom properties that may be specific to the contract at hand.

Table 4.3 maps the previously used vulnerability names to the names of the detectors used in Slither. The rest of the vulnerabilities are listed in Table 4.6, including their descriptions, impact severity and confidence level as given on their Github page [49]. The available informational and optimisation detectors are also listed in Table 4.7.

4.2.3 MythX

MythX is a smart contract security analysis platform developed by ConsenSys [50] that integrates several tools into a single analyser [51]. This platform is offered to subscribers through an online API and comes with both a free and paid subscription tier. Developers can either submit their code with a truffle plugin, through the Remix online IDE [52] or integrate with the API using their own custom tooling. Reported vulnerabilities are categorised according to ConsenSys's SWC registry [35]. The system will also notify users whenever an `AssertionFailed` logging event was triggered, thus providing an alternative to regular Solidity `assert` statements [53].

Our Name	Slither Name
Unauthorised Ether Withdrawal	Functions that send ether to arbitrary destinations
Unauthorised Self-destruct	Functions allowing anyone to destruct the contract
Unauthorised Delegatecall	Controlled delegatecall destination
Authorisation Through Origin	Dangerous usage of tx.origin
DoS with Failed Call	Multiple calls in a loop
DoS from Greedy State	Contracts that lock ether
Re-entrancy	Re-entrancy vulnerabilities (theft of ethers) Reentrancy vulnerabilities (no theft of ethers)
Unchecked Call Return Value	Unchecked send Unused return values

Table 4.5: Mapping between our vulnerability names and the closest Slither detector names.

Implementation Overview

The platform runs all given code through the following three security analysis tools:

- Maru, a proprietary static analysis tool [54],
- Harvey, a proprietary fuzzer [55], and
- Mythril, an open-source symbolic executioner [56].

All three analysis results are combined into a single report that indicates the location of every detected vulnerability, including a sequence of transactions to recreate the bug in the case of Harvey and Mythril. Mythril will be discussed in more detail under Section 4.2.6.

Security Properties

MythX checks properties defined according to the SWC registry [35] (also maintained by ConsenSyS), but restricts them based on the subscription tier currently held by the user. Table 4.8 lists the 10 vulnerabilities checked by the free tier of MythX and Table 4.9 lists the rest.

4.2.4 Echidna

Echidna is an open-source fuzzer for Ethereum smart contracts [57] developed by Trail of Bits and, like Slither, is also included in the Crytic Github app. One of the key differences between this tool and the others discussed, is that

Vulnerability	Impact	Confidence
Right-To-Left-Override control character is used	High	High
State variables shadowing	High	High
Functions allowing anyone to destruct the contract	High	High
Uninitialized state variables	High	High
Uninitialized storage variables	High	High
Functions that send ether to arbitrary destinations	High	Medium
Controlled delegatecall destination	High	Medium
Re-entrancy vulnerabilities (theft of ethers)	High	Medium
Incorrect ERC20 interfaces	Medium	High
Incorrect ERC721 interfaces	Medium	High
Dangerous strict equalities	Medium	High
Contracts that lock ether	Medium	High
State variables shadowing from abstract contracts	Medium	High
Constant functions changing the state	Medium	Medium
Reentrancy vulnerabilities (no theft of ethers)	Medium	Medium
Dangerous usage of tx.origin	Medium	Medium
Unchecked low-level calls	Medium	Medium
Unchecked send	Medium	Medium
Uninitialized local variables	Medium	Medium
Unused return values	Medium	Medium
Built-in symbol shadowing	Low	High
Local variables shadowing	Low	High
Constructor called not implemented	Low	High
Multiple calls in a loop	Low	Medium
Benign re-entrancy vulnerabilities	Low	Medium
Dangerous usage of block.timestamp	Low	Medium

Table 4.6: List of Slither's built-in vulnerability detectors.

it analyses custom invariants that are expressed as Solidity functions instead of checking through a list of predefined properties.

Implementation Overview

This tool is written in Haskell and tests EVM programs running on HEVM, which is an EVM also written in Haskell and designed for debugging [58]. The system generates a variable number of transactions to the contract under test before resetting the EVM state and testing another sequence of transactions. Transaction input data is generated based off analysis of the contract's ABI, along with additional heuristics and use of the coverage data to repeat calls.

Description	Type	Confidence
Assembly usage	Informational	High
Deprecated Solidity Standards	Informational	High
Un-indexed ERC20 event parameters	Informational	High
Low level calls	Informational	High
Conformance to Solidity naming conventions	Informational	High
If different pragma directives are used	Informational	High
Incorrect Solidity version (<0.4.24 or complex pragma)	Informational	High
Unused state variables	Informational	High
Conformance to numeric notation best practices	Informational	Medium
State variables that could be declared constant	Optimization	High
Public function that could be declared as external	Optimization	High

Table 4.7: List of Slither's built-in informational and optimisation detectors.

ID	Title
SWC-100	Function Default Visibility
SWC-101	Integer Overflow and Underflow
SWC-102	Outdated Compiler Version
SWC-103	Floating Pragma
SWC-104	Unchecked Call Return Value
SWC-105	Unprotected Ether Withdrawal
SWC-106	Unprotected SELFDESTRUCT Instruction
SWC-108	State Variable Default Visibility
SWC-110	Assert Violation
SWC-111	Use of Deprecated Solidity Functions

Table 4.8: List of SWC registry entries checked by the free tier of MythX.

Security Properties

Although Echidna does not test against any of the previously listed vulnerabilities natively, users are able to provide their own invariants to perform custom property analysis. By default the system will look for functions with the `echidna_` prefix and expect a boolean value indicating whether or not the property still holds.

Since these properties are defined within Solidity itself, you only have access to the same context that you would otherwise have in a regular Solidity function. If, for example, you want to check if an owner variable does not change after contract initialisation, then you can use a configuration file to define the exact address of the contract and write the following function:

```
function echidna_check_owner() returns (bool) {
    return owner == address(0xBf4eD7b27F1d666546E30D74d50d173d20bca754);
```

ID	Title
SWC-107	Reentrancy
SWC-109	Uninitialized Storage Pointer
SWC-112	Delegatecall to Untrusted Callee
SWC-113	DoS with Failed Call
SWC-114	Transaction Order Dependence
SWC-115	Authorization through tx.origin
SWC-116	Timestamp Dependence
SWC-117	Signature Malleability
SWC-118	Incorrect Constructor Name
SWC-119	Shadowing State Variables
SWC-120	Weak Sources of Randomness from Chain Attributes
SWC-121	Missing Protection against Signature Replay Attacks
SWC-122	Lack of Proper Signature Verification
SWC-123	Requirement Violation
SWC-124	Write to Arbitrary Storage Location
SWC-125	Incorrect Inheritance Order
SWC-127	Arbitrary Jump with Function Type Variable
SWC-128	Gas Exhaustion
SWC-129	Typographical Error
SWC-130	Right-To-Left-Override control character

Table 4.9: List of SWC registry entries checked by the premium tier of MythX.

```
}
```

The configuration file can also be used to set other parameters such as the available addresses to use when sending transactions or the initial Ether balance when deploying a contract. Users can make use of the API and Hedgehog library to conduct further property analysis using Haskell [59].

4.2.5 Manticore

Manticore is a symbolic analysis tool for Linux ELF binaries and the EVM [8] developed by Trail of Bits [47]. It features its own implementation of the EVM specification with the ability for instructions to operate on symbolic data. This tool features various built-in vulnerability detectors that will print detailed reports and provide a sequence of inputs to recreate the finding. Users may make use of a Python API to have more control over the execution environment setup and add their own vulnerability detectors. The tool is also made available through a command-line interface that analyses contracts symbolically up to a specified number of transactions [60].

Implementation Overview

The Manticore symbolic execution engine currently supports the Frontier version of the EVM specification. Manticore is able to model aspects of the blockchain environment symbolically (e.g. the timestamp or block number), along with each contract's storage, memory and Ether balance. Transactions can be sent with completely symbolic data (including symbolic gas) to a symbolic address, but the recipient will be deterministically concretised to all available accounts. States that are forked like in this case will be distributed to a task pool of available workers for parallel symbolic execution, since each state is given a complete copy of the simulated blockchain.

Vulnerability findings are reported during execution by using a type of Python object called a detector. These detectors listen for event callbacks that are triggered at certain points of execution, for example before and after every transaction or instruction. Each detector has access to the current system state, its own persistent context variable, and additional callback specific information such as the return value of a message. They are also able to taint variables for later analysis and concretise symbolic values using the system solver (Z3).

Security Properties

Although users are able to write and run their own detectors via the API, the Manticore command-line application comes prepackaged with a variety of built-in detectors. Table 4.10 includes a mapping between our vulnerability names and the names of each closest built-in Manticore detector. The full list of Manticore detectors are listed in Table 4.11.

4.2.6 Mythril

Mythril is a symbolic analysis tool for the EVM developed by ConsenSys. This tool automatically detects a set of security vulnerabilities in the given EVM bytecode or Solidity file and reports a sequence of inputs that recreates the exploit. The proprietary MythX security platform includes Mythril as part of its analysis process, but the publicly available Mythril Classic still remains open-source. It comes with both a command-line interface and Python API, with both having the option to generate state space graphs and pull concrete data from the blockchain for use in analysis.

Implementation Overview

Mythril symbolically analyses the given contract's CFG using its internal symbolic execution engine, Laser. Laser is able to simulate a sequence of transactions as specified by the user, with additional parameters available to control

Our Name	Manticore Name
Unauthorised Ether Withdrawal	Reachable external call or ether leak to sender or arbitrary address
Unauthorised Self-destruct	Reachable selfdestruct instructions
Unauthorised Delegatecall	Problematic uses of DELEGATECALL instruction
Authorisation Through Origin	Use of potentially unsafe/manipulable instructions
Re-entrancy	Reentrancy bug
Integer Overflow	Integer overflows
Unchecked Call Return Value	Unused internal transaction return values
Transaction Order Dependence	Possible transaction race conditions
Weak Sources of Randomness	Use of potentially unsafe/manipulable instructions
Assert Violations	Enable INVALID instruction detection
Ether Invariants	Use balance in EQ

Table 4.10: Mapping between our vulnerability names and the closest Manticore detector names.

the depth of the search and various solver timeout lengths. Transactions are created using a designated creator address and subsequent transactions are constrained to be sent from either the creator user account or one of two other user accounts in the environment.

Unlike Manticore, Mythril stores a copy of the symbolic call graph that can be used to explore states for vulnerabilities or exported to, for example, a graphical format that can be used to help with debugging.

Security Properties

Table 4.12 shows all the SWC registry vulnerabilities checked by Mythril's built-in analysis modules. The analysis modules can all be configured to be called after specific opcodes are reached in the symbolic execution engine so that they can analyse the state space and EVM data to potentially warn the user about a vulnerability. Several vulnerabilities, for example re-entrancy, that are not checked by the free version of MythX are included in the Mythril package distribution.

Description	Impact	Confidence
Problematic uses of DELEGATECALL instruction	High	High
Reentrancy bug	High	High
Reentrancy bug (different method)	High	High
Integer overflows	High	High
Use balance in EQ	High	High
Use of potentially unsafe/manipulable instructions	Medium	High
Reachable selfdestruct instructions	Medium	High
Reachable external call or ether leak to sender or arbitrary address	Medium	High
Uninitialized memory usage	Medium	High
Uninitialized storage usage	Medium	High
Enable INVALID instruction detection	Low	High
Unused internal transaction return values	Low	High
Possible transaction race conditions	Low	Low

Table 4.11: List of built-in Manticore detectors.

ID	Title
SWC-101	Integer Overflow and Underflow
SWC-104	Unchecked Call Return Value
SWC-105	Unprotected Ether Withdrawal
SWC-106	Unprotected SELFDESTRUCT Instruction
SWC-107	Reentrancy
SWC-110	Assert Violation
SWC-111	Use of Deprecated Solidity Functions
SWC-112	Delegatecall to Untrusted Callee
SWC-113	DoS with Failed Call
SWC-116	Timestamp Dependence
SWC-120	Weak Sources of Randomness from Chain Attributes
SWC-127	Arbitrary Jump with Function Type Variable

Table 4.12: List of SWC registry entries checked by Mythril Classic.

4.3 Other Tools

In this section we discuss some of the more notable tools in the Ethereum community that we have omitted from the previous discussion. Many of these tools implement advanced analysis techniques that could make its way into newly developed analysis frameworks.

Oyente [61] was one of the first symbolic execution tools implemented

for the EVM and used to analyse thousands of smart contracts from the blockchain. Similarly, Nikolić et al. developed a tool called MAIAN [38] that analysed on-chain contracts for trace vulnerabilities (i.e. those that span multiple transactions). Kolluri et al. developed EthRacer to symbolically analyse bytecode for cases of transaction order dependence [62]. We also note the research done by Krupp and Rossow to develop teEther [63] that symbolically executes a given contract's CFG to find vulnerabilities such as an unauthorised self-destruct, delegatecall or withdrawal.

Jiang et al. developed one of the earliest fuzzing tools for smart contracts called ContractFuzzer [64] that is able to detect multiple vulnerability types using a set of instrumented oracles within their framework. Input is generated according to the contract's ABI. Although not directly associated with finding vulnerabilities in smart contracts, Fu et al. implemented EVMFuzzer [65] to find vulnerabilities and inconsistent behaviour in several of the of most popular EVM clients such as geth and aleth.

Several static analysis tools are available for Solidity smart contracts that warn about potential vulnerabilities and bad coding practices. This includes tools such as Smartcheck [66], Octopus [67], Solhint [68], and the Remix IDE's analysis component [6].

In 2018 Brent et al. introduced the Vandal static analysis framework [69] that allows users to specify vulnerability properties using Soufflé. Other tools that can formally verify and allow users to reason about vulnerabilities using correctness properties include VerX [70], VeriSolid [71], and a formal semantic implementation of the EVM using the K framework [72].

Lastly, we note that there are a variety of other tools that focus on analysing gas-based properties and vulnerabilities within Ethereum smart contracts. Tools such as GASPER [73] automatically locate inefficient gas patterns in code, while MadMax [74] analyses the code for vulnerabilities relating to its gas usage.

Chapter 5

Extensions

We have thus far considered a variety of tools that analyse properties using different methodologies, but based on our vulnerability classification, the only tool that came close to achieving full coverage was the proprietary version of MythX that combines the results of multiple tools. In this chapter we investigate how one can use the official APIs supported by these tools, specifically Manticore, to analyse more vulnerabilities and improve code coverage.

5.1 Framework Architecture

Although Manticore does support a command-line interface as mentioned in Chapter 4, this method of analysing contracts does not provide the same flexibility in setting up the execution environment as one would have when using its Python API.

Some contracts are only exploitable after a certain amount of time has elapsed or at specific block numbers. They may also require interaction with other smart contracts before a vulnerable state is reached, for example in the case of re-entrancy. An ideal security analysis tool should be able to correctly model these environmental characteristics, alert the user of any potential vulnerabilities, and produce a set of instructions (including malicious code) to recreate these exploits.

Our analysis framework is therefore structured in a pipeline format, since all the main components must be used in the following sequential order:

1. Setting up the analysis environment;
2. Performing symbolic analysis;
3. Validating and preparing results.

Figure 5.1 shows the architecture of this symbolic execution pipeline. Each listed component will be discussed in more depth in Section 5.2.

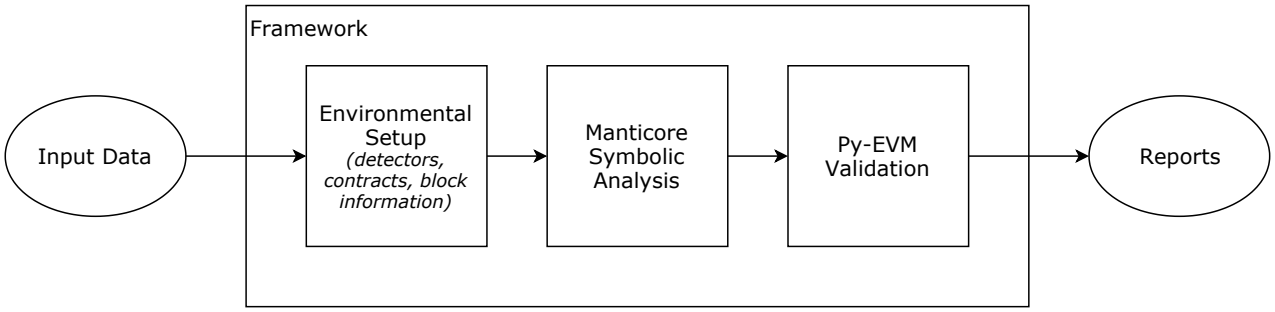

Figure 5.1: Structure of the symbolic execution framework outlining the major components.

5.2 Framework Implementation

Each of the following subsections explain a component of the framework in further detail.

5.2.1 External Interfaces

The framework is intended to be available both as a command-line interface (CLI) and Python API, similar to the base Manticore system. While the CLI provides the configuration options necessary to analyse the majority of smart contracts, some users may require additional configuration before running analysis (e.g. deploy other smart contracts) by instantiating the system within a custom Python script.

Regardless of the method used to interface with the system, the following list of input arguments are available:

- The Solidity contract under test;
- The number of transactions to analyse symbolically;
- The path to the file in which results should be saved;
- An option to only simulate transactions from a single attacker address;
- An optional timeout length;
- Which vulnerability detector modules to activate.

Usage of every detector will increase analysis time significantly and can be safely omitted if the goal is to only find specific vulnerabilities (e.g. assertion errors).

5.2.2 Environmental Setup

The analysis component of the system will take several steps to instantiate the main Manticore execution environment and configure it accordingly.

One of the first steps taken is to analyse the given smart contract file using Slither so that the system can statically determine the name of each contract listed within, whether or not they accept Ether in their constructor functions, and the name of each publicly accessible function. Function names may contain special prefixes to indicate that they serve as oracle functions that test some invariant, namely:

- "`manticore_`" to return `true` whenever violated or
- "`test_`" to return `false` in case an invariant property is violated.

Two user accounts are initially created, one for the owner of the main contract under test and one malicious actor, each with a balance of 1000 Ether to ensure that they can afford to pay for transactions.

Each configured detector is initialised, along with a symbolic block number and timestamp, before the system makes the first transaction: creation of the specified contract. The contract itself is owned by the owner account defined previously, is given a symbolic Ether balance if capable of receiving Ether, and is passed symbolic arguments. If successfully instantiated, then the system can continue with setting up an attacking contract in the case where the user activated the re-entrancy detection module.

Attacking Contract Setup and Synthesis

Re-entrancy exploits always involve at least one adversarial (or attacking) smart contract in the system that is able to make re-entrant calls to its target contract, as discussed previously in Section 3.2.3. For us to provide the user with a concrete example of the entire re-entrancy exploit, we need to synthesise these attacking smart contracts and provide a sequence of concrete transactions to recreate the loss of Ether funds. This synthesis of the attacking contract is similar to that of the SmartScopy [75] tool, however, due to time constraints we have not been able to implement any of their early pruning strategies in the Manticore symbolic execution environment.

Our approach instead loads a contract that contains a skeleton structure of a re-entrant attacking contract, where the function content (i.e. how to respond to an incoming message call) are all left symbolic up to a predefined bound. The full version of this contract is included in Appendix A on page 98 and is based off the `GenericReentranceExploit` contract included in the Manticore repository's list of examples [60].

Listing 14 shows the `proxy` function included within the attacking contract. Its goal is to start the initial call to the contract under test — stored in the

`vulnerableContract` field — and pass a symbolic Ether value and data buffer. These symbolic values are set beforehand for each of the main transactions, specified by the `txCounter` variable that increments with each proxy call. In practice, this allows the system to symbolically execute each function in the target contract with the attacking contract as the caller. If the target

```
1 function proxy() public payable {
2     vulnerableContract.call.value(msg.value)(proxyData[txCounter]);
3     txCounter += 1;
4 }
```

Listing 14: Attacking contract proxy function.

contract sends any Ether back to the attacking contract, for example during a withdrawal process, then the default function in Listing 15 will be executed. Here we have two potential outcomes, depending on the type of call:

1. If the call was made to this contract using a non-empty string of data, then the attacking contract will return a 32 byte string of data (lines 3 to 8);

2. Otherwise, the contract will call the target contract again using symbolic data up to a maximum number of times defined by the `counter` variable (lines 10 to 13).

The goal of the first option is to mimic any unknown function that expects a return value, since that function's signature would in practice not be implemented by the attacking contract and will be handled by the default function. This allows the attacking contract to respond to regular function calls with symbolic return values and potentially improve coverage.

If it is a regular default function call, for example just a transfer of Ether, then this contract will simply attempt to make a re-entrant call back into the target contract. The symbolic values of these re-entrant calls are set beforehand for each `counter` value.

5.2.3 Vulnerability Detectors

Detectors are Manticore objects that listen to a variety of callbacks and potentially add a finding at a specified position in the code. Callback events are fired before and after for example:

- The execution of an EVM instruction;

- Writing to a contract's storage;

```
function () payable {
    if (msg.data.length > 0) {
        bytes32 data = returnData[txCounter];
        assembly {
            let r := data
            mstore(0x0, r)
            return(0x0, 32)
        }
    } else if (counter > 0) {
        counter -= 1;
        vulnerableContract.call.value(symbolicValues[counter])(
            symbolicData[counter]
        );
    }
}
```

Listing 15: Attacking contract default function.

- A transaction is opened or closed.

State information related to the event (values written or returned) is passed to the detector, at which point it can even concretise symbolic values as part of its evaluation. Listing 16 shows an example of one of these callback events implemented by the built-in Manticore self-destruct detector. The function is passed a `state` variable containing information related to the symbolic system state and EVM world (e.g. current path constraints, account balances), and two other variables with information about the EVM instruction that is about to be executed: `instruction` and `arguments`. If a vulnerable state is reached, then the system can flag that state for later reporting using the `add_finding_here` method.

The rest of this section lists each vulnerability detected by this framework and explains how the corresponding detector was implemented in more detail.

```
def will_evm_execute_instruction_callback(self, state,
                                          instruction, arguments):
    if instruction.semantics == "SELFDESTRUCT":
        self.add_finding_here(state, "Reachable SELFDESTRUCT")
```

Listing 16: Self-destruct detector function handling EVM instruction callbacks.

Unauthorised Ether Withdrawal

Manticore's built-in unauthorised Ether withdrawal detector is invoked after every `CALL` instruction and warns whenever a call is made to the message sender (`msg.sender`) or if the recipient address is an arbitrary symbolic address. Addresses are considered arbitrary if they can be solved to more than one value.

A similar detector was implemented that checks if the recipient can be arbitrary, but also warn if the contract's entire Ether balance can be withdrawn by an address that did not create the contract.

Unauthorised Self-destruct

We extended the Manticore self-destruct detector to add additional information depending on the recipient of the self-destruct. The base version (listed partially in Listing 16) simply warns whenever a self-destruct instruction is reached, but we can symbolically evaluate the recipient value to determine if it is arbitrary and warn accordingly.

Unauthorised Delegatecall

The built-in Manticore detector for `DELEGATECALL` instructions is used to detect this vulnerability. It warns whenever the recipient of the `DELEGATECALL` is an arbitrary symbolic value and if the function selector of the call (i.e. the first four bytes that determine the type of function) can also be arbitrarily controlled by the caller.

DoS from Greedy State

Contracts suffer from this type of denial of service whenever it is impossible for them to send Ether, while still being able to receive it. In other words, there is no way for a user to withdraw their funds, even if they force the transfer of funds through a self-destruct with this contract as the beneficiary. Since we can not explore every possible state of the system using our symbolic engine, we need to approximate this vulnerability detection by only warning if every terminal node from a given initial state never sent a non-zero Ether value.

Manticore does not yet store the symbolic execution tree in a traversable format. We therefore extend the system by storing the ID of each state's parent and then build up a tree structure while executing the program. Our detector then marks all initial states in each transaction run, as well as those that send Ether. At the end of the run we perform a depth-first search for an Ether sending node on each initial state and warn if no instances are found.

Re-entrancy

Two different re-entrancy detectors are registered before the state of execution, the first of which comes packaged with Manticore and warns of re-entrancy whenever storage is written after a flagged call. Calls can be flagged if they have a user-supplied address as recipient, which we set as the attacking contract if the user requests a check for re-entrancy.

We implemented another re-entrancy detector to check whether Ether can be stolen by the attacker. Our detector saves the expected Ether balance of the contract initiating the message call and compares it with the balance after the call was completed. If that value (generally a symbolic one) can evaluate to a value less than the expected value, then we know that the recipient of the call somehow re-entered the contract and withdrew additional Ether.

Integer Overflow

Manticore's integer overflow detector checks for overflow by doubling the bit-size of the operands to 512 bits and then checks if the result falls outside of the 256 bit minimum or maximum value. This check is performed for unsigned and signed addition, subtraction, and multiplication.

Write to Arbitrary Storage Locations

We present two approaches to detecting this vulnerability. The first is to implement a check before each storage write instruction (`SSTORE`) to see if the offset position is a symbolic value and if it can evaluate to an arbitrary value.

Our other approach is to take a user-supplied storage offset position and warn if the value at that position is ever overwritten during execution. A good candidate for this use case would be an owner address or some other critical variable that is set during contract creation, but is intended to remain immutable.

Ether Invariants

For this type of vulnerability we again require the user to supply an offset position in storage so that it can be compared with the contract's Ether balance at the end of every transaction. If this value differs from that of the Ether balance, then the invariant is violated and a warning is raised.

We also implemented a detector to warn whenever a transaction closed in a state where a contract had to forcefully receive Ether. Contracts can forcefully receive Ether via self-destructs without triggering any contract code, which we simulate by adding symbolic Ether amounts to the contract before each transaction (as explained later in Section 5.2.4). A warning is raised to

the user if it is infeasible for this symbolic value to evaluate to zero after the current transaction (i.e. the contract *must* receive Ether somehow).

Unchecked Call Return Value

The base Manticore unchecked call return value detector is registered by our framework. This detector taints all call return values pushed to the EVM stack and checks that they were used in its control flow at the end of a transaction.

Transaction Order Dependence

This vulnerability is checked using the Manticore detector, where it taints values written to storage and warns when they are used in subsequent transactions. It also saves the name of the functions that were called to help display a more informative message when adding the finding.

Weak Sources of Randomness and Environmental Variables

We use the Manticore detector that warns whenever an environmental instruction, for example `TIMESTAMP`, `NUMBER`, and `ORIGIN`, is used. It is currently up to the user to read the finding reports and determine whether or not this instruction usage can be considered dangerous or manipulable by attackers.

Assert Violations

Assertion violations are handled similar to the previous detector, where Manticore simply notifies the user whenever an `INVALID` instruction was reached. We also added an extension to warn whenever an `AssertionFailed` logging event is emitted, similar to the ones implemented in MythX [53].

Custom Properties

We allow the use of custom properties implemented as Python expressions that allow the user to check for program invariant violations after every call. These properties are given access to a variable called `balance` that represents the current contract's balance, as well as a dictionary called `storage` that likewise represents the current contract's persistent storage. For example, the following property:

```
storage[0] == (storage[1] + storage[2])
```

will be symbolically evaluated after calls to determine if the value storage offset `0` is the same as the sum of offsets `1` and `2`. If this expression can symbolically evaluate to false, then a warning is raised. Potential future work for this feature is discussed in Section 5.3.

5.2.4 Execution

Once all the contracts are deployed and detectors initialised, the system can start analysing the specified smart contract symbolically. The system will:

1. Symbolically increase the block number and timestamp to simulate the passage of time;
2. Force a symbolic amount of Ether to the contract to simulate the receiving of Ether from some other contract's self-destruct;
3. Send a transaction with symbolic data to the contract.

This will continue until the maximum number of transactions is reached, the system times out, the execution engine does not have any states left to analyse, or if coverage does not improve between runs (if configured explicitly). Manticore also has a built-in option to stop further exploration of states that do not affect the world state (e.g. write to storage, revert), thus saving some execution time by pruning unnecessary states.

Block numbers and timestamps are not allowed to be increased by more than a week at a time, thus excluding some infeasible waiting periods (e.g. waiting years for the timestamp to overflow). The timestamp is also constrained to increase at a minimum of 5 seconds between transactions. Lastly, the block number and timestamp are both constrained to either remain equal between transactions (indicating that they belong to the same block) or to increase between transactions and thus separate them.

Transaction data byte arrays and Ether values are symbolic, however, the target address and calling address depend on some of the configuration options. The system can be configured to only send transactions as the attacking user account and none as the creator, thereby simulating a scenario where the owner seldom or never interacts with their contract. If this option is not enabled, then transactions can be sent by either and will naturally increase execution time as more states are explored.

When re-entrancy detection is enabled, transactions sent by the attacker user account will be constrained to only interact with the target address via the attacking contract's `proxy` function. The target contract will therefore only see messages from either its creator or the attacking contract.

5.2.5 Validation

Our final step in the analysis pipeline is to validate the sequence of concrete transactions produced as output by Manticore on a test network. The framework makes use of the Ethereum Tester Python library [76] to run a local instance of Py-EVM [21].

For each given run of transaction calls, the system must create all applicable user accounts, load the contracts with their concrete arguments, and finally send each transaction. The status of the Py-EVM transaction is compared with that of the Manticore result and marked whether or not it is valid in the generated report.

5.2.6 Output

The framework outputs largely the same information as that of Manticore in a JSON format, most important of which is:

- The coverage of analysis of contract code;
- The number of transactions called and original limit;
- The number of states analysed internally by Manticore;
- Timing information and whether or not it timed out;
- A list of vulnerability findings and their associated locations in the code.

These findings also include concrete transaction and environmental data that allows the user to recreate the detected vulnerability. Regular Manticore reporting and finalisation steps can also be activated, but it comes at a cost given how it concretises every terminated end state, instead of just those that contain findings.

5.3 Limitations and Future Work

Our attacking contract synthesiser can be expanded to cover more complex external contract interactions, such as those involving malicious `DELEGATECALL` instructions or re-entrancy during contract creation [77]. However, additional heuristics may have to be implemented to optimise calls to our attacking contract before it is expanded, since it causes a significant increase in execution time.

The custom property support in our framework can be expanded to allow more control over when the properties are evaluated (after e.g. storage writes, calls, transactions, etc.) and to allow previous states to be referenced as part of the property expression. With these expansions it might also be necessary to implement a DSL for custom properties, similar to that of VerX [70].

Lastly, the framework can be further extended to integrate with truffle or other analysis tools, similar to how Manticore is currently included in the Etheno security analysis suite [78]. Truffle could be used to gather contract compilation artefacts or manage imports, as well as specify the contract's

deployment procedures. If a contract's deployment requires other prerequisite contracts before it can function correctly, then this will help avoid unnecessary analysis if added in the tool's environmental setup.

Chapter 6

Evaluation

In this chapter we demonstrate how the tools discussed in Chapters 4 and 5 can be used to find vulnerabilities in smart contracts and we evaluate their effectiveness and efficiency in doing so.

We first discuss the setup of our tools and environment in Section 6.1, since some of them require additional configuration before they are able to successfully evaluate certain examples.

Next we evaluate the tools as part of three separate experiments in Sections 6.2, 6.3 and 6.4. These experiments evaluate the tools' effectiveness in automatically detecting vulnerabilities, detecting program invariants that indicate exploitation, and overall coverage performance, respectively.

Lastly, we compare the results of each experiment and discuss our overall findings in Section 6.5.

6.1 Setup

Our evaluation over the course of the next few sections will only consider the main tools highlighted during the previous chapters:

Securify The web version accessible at the time of writing.

Slither Version 0.6.6 of the command-line application.

MythX The free subscription tier of analysis accessible at the time of writing (available through the Remix IDE plugin).

Echidna A branched version of version 1.1.0.0 that allows some of the generated transaction inputs to send Ether.

Mythril Version 0.21.20 of the command-line application in its standard configuration and with a small modification to only send transactions from a single user account.

Manticore Our framework as described in Chapter 5, hereafter referred to as "Manticore+". Its extended version of Manticore is based off of version 0.3.2.

Each of the evaluation sections will list the tools analysed in the experiment and any additional configurations or modifications of standard behaviour specific to that experiment.

Note that all command-line applications (except for Slither) were executed and evaluated on an IBM System x3650 M4 server with:

- Two Intel Xeon E5-2640 v2 CPUs (16 cores and 32 threads total);
- 283 GB RAM; and
- 4.5 TB of mixed usable storage.

6.2 Vulnerability Reporting Evaluation

The aim of this first experiment is to evaluate the effectiveness of each tool when used to automatically detect vulnerabilities and predefined properties in a given smart contract. First, a subset of the vulnerabilities analysed in Chapter 3 will be chosen for evaluation, along with the tools that are able to automatically report their violations. Vulnerabilities that have too much room for interpretation when it comes to the detection criteria and tool-specific implementation, will be omitted for the sake of fairness.

Each tool will then be evaluated against a large selection of contracts that are each guaranteed to contain at least one instance of a particular vulnerability and measured to count the number of successfully reported findings.

6.2.1 Contracts

Table 6.1 lists each of the vulnerabilities chosen and their number of unique occurrences across all the included contracts. These contracts were gathered from various articles and online repositories, including:

- The ConsenSys smart contract weakness classification registry [35];
- The Trail of Bits "Not so smart contracts" repository [79];
- Online smart contract security challenges [80], [81];
- Various other sources with examples of vulnerabilities [32], [38], [60], [82], [83].

Vulnerability	Contracts
Unauthorised Ether Withdrawal	8
Unauthorised Self-destruct	5
Re-entrancy	7
Integer Overflow	12
Assert Violations	13
Unchecked Call Return Value	3

Table 6.1: List of vulnerabilities chosen for evaluation and the number of vulnerability occurrences included in each test suite.

6.2.2 Tools

All the tools discussed in Section 6.1, except Echidna, are evaluated against these contracts. Echidna requires the user to provide oracle functions to test program invariants, which will be the topic of the next experiment. Mythril and Manticore+ are evaluated up to three symbolic transactions and given a timeout of 30 minutes.

Not all of the tools are capable of detecting every vulnerability and will therefore not be evaluated against those contracts:

- Securify and Slither do not detect integer overflows or assertion violations;
- The free tier of MythX does not detect re-entrancy.

6.2.3 Results

The results for each vulnerability category and tool is given in Table 6.2, where each cell indicates the number of successful reported test cases. There are several notable differences in the number of findings reported among the tools:

1. Slither reports far fewer instances of an unauthorised Ether withdrawal than Securify, suggesting that its criteria for what constitutes unauthorised access may differ from that of the other tools.
2. Securify and Slither reported the same number of unauthorised self-destructs, which was significantly less than that of the other tools and suggests once again different criteria.
3. Securify and Slither both reported more instances of re-entrancy than the two symbolic tools, but in this case it is important to note that Manticore+ and Mythril will only report for feasible paths reached within its bounded execution.

4. Mythril outperformed Manticore+ in searching for assertion violations, suggesting that it may have better overall path coverage under these execution constraints.

5. Mythril outscored or equalled MythX in every category, most likely due to the fact that MythX was capped at a two minute execution time at this free subscription tier. A more in-depth comparison between each tool within the MythX suite can only be done with access to all the individual tools.

The set of sample contracts in each test case only includes contracts that are known to contain an instance of that particular vulnerability. Determining the false positive reporting rate for these tools would therefore require us to include a fair number of additional samples that are guaranteed to not contain the vulnerability and measure the number of false reports.

Determining the false positive rate for a tool like Securify under these conditions would help determine whether or not the time saved in automated analysis is worth the time spent reviewing potentially false reports. Based on this selection of contracts, however, we note that Securify and MythX are able to report nearly all re-entrancy, overflow and assert violations. Manticore+ and Mythril missed at least one of these example re-entrant contracts due to a lack of coverage in that particular scenario.

Vulnerability	Contracts	Securify	Slither	MythX	Manticore+	Mythril
Unauthorised Ether Withdrawal	8	8	2	6	7	7
Unauthorised Self-destruct	5	2	2	4	5	5
Re-entrancy	7	6	6	N/A	5	4
Integer Overflow	12	N/A	N/A	10	11	11
Assert Violations	13	N/A	N/A	11	9	12
Unchecked Call Return Value	3	3	3	3	3	3

Table 6.2: Results for the vulnerability detector evaluation. The number of successful detections is shown for each tool compared to the total number of contracts in each respective data set.

6.3 Challenge Contract Evaluation

Our next experiment considers a set of contracts provided as part of the Capture the Ether [81] and Ethernaut [80] challenges that aim to test the effectiveness of our analysis tools in finding different types of exploits. Both of these challenges are hosted online as interactive games where players deploy and exploit contracts on a test Ethereum network, with each level representing a different contract. Players may use all available features and live contracts

on the test network to beat levels, including custom attacking smart contracts written specifically to exploit a vulnerability.

These contracts were chosen as benchmarks for further effectiveness evaluation, since they contain oracle functions that indicate whether or not the contract was successfully exploited. The Capture the Ether set simply includes a function, `isComplete`, within every contract that returns `true` when the level is beaten. Ethernaut include a validation function as part of its test harness (accessible at its GitHub repository [84]) and performs a similar test to determine if the contract was successfully exploited. We included the Ethernaut test harness setup and validation functions in the challenge contract itself, in order to make it easier to compare it to Capture the Ether.

There are full examples on how to complete the levels available for each exploit, both in the source documentation found on GitHub in the case of Ethernaut, and in the guide to Capture the Ether, published by Enigmatic [85].

6.3.1 Contracts

Table 6.3 and Table 6.4 contains a list of the Capture the Ether and Ethernaut contract data sets respectively. Each table entry lists the number of contracts, functions, and lines of code included in the smart contract source file provided.

6.3.2 Tools

For this experiment we only compare the tools that are able to report on any occurrences of an invariant violation (i.e. our oracle functions) and generate sequences of transaction input data that recreate these exploits. The contracts are therefore tested using Echidna, Manticore+, and Mythril (the versions described in Section 6.1) and set to only send transactions from a single attacker address, since the contract owners play no part in these online challenges. Allowing the contract owner to make calls would in some cases trivialise the sequence, for example enabling full withdrawals or transfer of ownership.

6.3.3 Results

Tables 6.5 and 6.6 show the results for Capture the Ether and Ethernaut respectively, where empty columns indicate that the tool was not able to successfully analyse the given contract. Manticore+ and Mythril were both allowed to analyse up to three transactions, except in the cases that required more transactions to find the exploit as specified in Tables 6.3 and 6.4 (with a timeout of 3 hours). Both symbolic tools successfully exploit significantly more contracts in the first set than Echidna, most of which require very specific pseudo-random number inputs and data that triggers integer overflow.

ID	Name	C	F	LoC	T
A1	Guess the Number	1	2	16	1
A2	Guess the Secret Number	1	2	16	1
A3	Guess the Random Number	1	2	17	1
A4	Guess the New Number	1	2	16	1
A5	Predict the Future	1	3	28	2
A6	Predict the Block Hash	1	3	28	2
A7	Token Sale	1	3	20	2
A8	Token Whale	1	5	41	3
A9	Retirement Fund	1	3	29	1
A10	Mapping	1	2	14	1
A11	Donation	1	3	28	2
A12	Fifty Years	1	3	41	6
A13	Fuzzy Identity	1	4	28	1
A14	Public Key	1	1	9	1
A15	Account Takeover	1	1	9	1
A16	Assume Ownership	1	2	12	2
A17	Token Bank	2	9	81	2

Table 6.3: List of benchmarks in the Capture the Ether data set. Includes the number of contracts (C), functions (F), and lines of code (LoC) contained in each file. The approximate number of transactions (T) required to complete the level is also shown.

The second set of contracts contains a more diverse group of vulnerabilities and lead to an improvement in Echidna's overall effectiveness. Mythril was able to exploit the most contracts out of the two symbolic tools (20 vs 17), but if the Echidna results are also considered, then all three tools were able to analyse 24 out of the 39 contracts.

6.4 Performance Evaluation

Our last experiment aims to analyse the efficiency and general performance of the selected tools when applied to contracts that are in use on the main public Ethereum network. These contracts are generally larger than the ones analysed in the previous section and therefore model regular development environments more closely.

6.4.1 Contracts

For this experiment we evaluate the performance of Mythril and Manticore+ when testing the top 20 ERC-20 contracts listed on Etherscan [86] as of the

ID	Name	C	F	LoC	T
B1	Fallback	2	9	64	2
B2	Fallout	2	10	61	1
B3	Coin Flip	1	6	55	10
B4	Telephone	1	1	12	1
B5	Token	1	2	17	1
B6	Delegation	2	2	23	1
B7	Force	1	0	2	0
B8	Vault	1	1	14	1
B9	King	2	1	25	1
B10	Reentrancy	1	9	49	2
B11	Elevator	1	2	15	1
B12	Privacy	1	1	16	1
B13	Gatekeeper One	1	6	52	1
B14	Gatekeeper Two	1	1	22	1
B15	Naught Coin	5	20	123	1
B16	Preservation	2	3	25	2
B17	Locked	1	1	18	1
B18	Recovery	2	9	55	1
B19	Magic Number	1	1	8	1
B20	Alien Codex	2	4	32	3
B21	Denial	1	9	51	1
B22	Shop	1	2	15	1

Table 6.4: List of benchmarks in the Ethernaut data set. Includes the number of contracts (C), functions (F), and lines of code (LoC) contained in each file. The approximate number of transactions (T) required to complete the level is also shown.

time of writing. Table 6.7 shows all the contracts included in this experiment, as well as the size of the bytecode used to initialise these contracts as shown on Etherscan's page for each token.

These contracts are generally larger than the average ones analysed in the previous experiment. The additional functions increase the number of possible program paths that need to be followed during each transaction and thus decrease the overall number of successive transactions that can be analysed in a fixed amount of time. Concrete constructor arguments are included with the Etherscan initialisation bytecode and are passed as is to Mythril during analysis. Manticore+ can only accept a Solidity file and instead attempts to compile with Solidity version 0.4.25 and pass symbolic constructor arguments.

	Oracle			Time (s)			Coverage (%)	
ID	Mythril	Manticore+	Echidna	Mythril	Manticore+	Echidna	Mythril	Manticore+
A1	Yes	Yes		14	14	37	99	95
A2				9	17	37	76	65
A3	Yes	Yes		14	15	36	99	95
A4	Yes	Yes		15	785	37	99	96
A5	Yes	Yes		29	13288	42	99	97
A6	Yes	Yes		28	26203	43	99	96
A7				129	2565	58	99	97
A8				9074	10800	100	99	92
A9	Yes	Yes		16	41	39	63	64
A10	Yes	Yes		178	1818	57	99	99
A11	Yes	Yes		115	1680	52	99	98
A12	Yes	Yes		10800	10800	2552	99	96
A13				10	4	40	59	34
A14	Yes			689	11	70	99	71
A15				8	3	35	79	64
A16	Yes	Yes	Yes	18	11	6	99	99
A17					838	112		81

Table 6.5: Capture the Ether benchmark results for each tool.

6.4.2 Tools

This experiment will analyse the performance of Mythril and Manticore+. It is configured to analyse the given contract and report the number of detected issues, without any restrictions on the account that sends the transactions. Re-entrancy detection in Manticore+ using the attacking contract module described in Section 5.2.2 is deactivated for the sake of comparison with Mythril, due to its significant increase in the number of program paths explored.

6.4.3 Results

Table 6.8 includes the results for Mythril when run with up to three symbolic transactions, showing the total number of errors reported, coverage percentage, and the execution time when limited incrementally up to a depth of three symbolic transactions. T1, T3, T7, T8, and T13 failed to initialise the given contract within the allocated recursive depth (50) and therefore only show the partial coverage of the initialisation bytecode.

Of the other 15 contracts that managed to initialise correctly, only 9 increased their coverage after the second transaction and none of them managed to improve after the third. This lack of additional coverage after three transactions comes at a significant cost in execution time, with six contracts reaching Mythril's default execution timeout of 24 hours.

	Oracle			Time (s)			Coverage (%)	
ID	Mythril	Manticore+	Echidna	Mythril	Manticore+	Echidna	Mythril	Manticore+
B1			Yes	95	328	35	99	90
B2	Yes	Yes	Yes	196	40	2	99	97
B3			Yes	10800	10800	75	98	89
B4	Yes			16	14	45	99	77
B5	Yes	Yes	Yes	31	64	7	98	98
B6				12	15	57	98	85
B7		Yes		7	6		98	98
B8	Yes	Yes		16	8	44	99	99
B9	Yes			140	646	50	99	90
B10				190	429	66	99	93
B11				22	18	51	71	53
B12	Yes	Yes		36	15	45	99	99
B13				83	15	44	98	47
B14				23	14	44	99	54
B15		Yes	Yes		1889	44		97
B16				21	199	68	34	93
B17	Yes	Yes	Yes	31	30	6	99	99
B18								
B19				54	20		95	70
B20	Yes			801	10800	63	99	65
B21	Yes			86512	1836		99	92
B22				30	17	46	98	48

Table 6.6: Ethernaut benchmark results for each tool.

Table 6.9 shows the results for Manticore+ when analysing the same contracts with a single symbolic transaction and a two hour timeout. The only contract that managed to successfully analyse the second transaction within a four hour timeout window was T18, where it took 3 hours and 15 minutes to complete and increased coverage to 99.97%. Coverage performance was similar compared to Mythril in the overlapping contracts, but execution time increased much more rapidly during the second transaction.

6.5 Discussion

Our first experiment compared the ability of each tool to report a number of different vulnerabilities contained within a set of contracts sourced from online repositories and articles. Each evaluated contract was guaranteed to contain one vulnerability in particular, according to its source, and the tools were evaluated simply based on whether or not that vulnerability was reported.

Securify and Mythx, two tools that typically perform analysis in less than

ID	Name	Bytes	LoC	Market Cap
T1	Tether USD (USDT)	23,801	251	$4,132,324,825
T2	BNB (BNB)	12,209	112	$2,252,784,046
T3	Bitfinex LEO Token (LEO)	17,171	339	$879,390,793
T4	ChainLink Token (LINK)	6,451	159	$735,352,870
T5	HuobiToken (HT)	3,113	77	$657,296,056
T6	Maker (MKR)	7,141	236	$496,970,792
T7	USD Coin (USDC)	5,779	109	$477,072,506
T8	Crypto.com Coin (CRO)	20,941	323	$369,224,636
T9	HedgeTrade (HEDG)	15,281	245	$353,066,642
T10	VeChain (VEN)	9,165	358	$315,905,856
T11	Ino Coin (INO)	5,929	71	$283,290,728
T12	BAT (BAT)	7,727	129	$262,909,359
T13	Paxos Standard (PAX)	4,187	104	$237,558,126
T14	Synthetix Network Token (SNX)	3,351	156	$210,540,663
T15	Insight Chain (INB)	13,385	151	$182,537,127
T16	TrueUSD (TUSD)	3,175	94	$160,650,286
T17	Centrality Token (CENNZ)	10,587	97	$133,670,012
T18	KaratBank Coin (KBC)	6,927	142	$128,254,150
T19	ZRX (ZRX)	6,125	76	$125,880,783
T20	Reputation (REP)	1,659	504	$111,969,417

Table 6.7: Top 20 ERC-20 tokens according to their market cap (volume times approximate price in USD) on 2019/12/15. The length of the contract initialisation bytecode (bytes), and lines of code (LoC) in Solidity is also shown.

10 minutes, tended to score relatively well compared to their symbolic execution counterparts (Manticore+ and Mythril). Securify outscored both in re-entrancy and unauthorised Ether withdrawal, whereas MythX only failed to detect one of Mythril's contracts in overflow and assertion violations. Manticore+ and Mythril performed equally well in four of the categories, with the former scoring one better in re-entrancy and the latter catching more assertion violations.

The second experiment evaluated three tools (Echidna, Manticore+, and Mythril) against a set of contracts meant to be exploited as part of an online challenge. Both symbolic tools performed much better than Echidna in the Capture the Ether data set, due to those contracts mostly being comprised of integer overflow exploits and pseudo-random number guessing games.

In the Ethernaut set, the two symbolic executioners struggled to maintain the same level of performance. These contracts cover a wider array of vulnerabilities, as well as more of an emphasis on interactions between contracts that neither the base symbolic tools nor our own framework are able to support.

	Coverage (%)			Errors			Time (minutes)		
ID	TX1	TX2	TX3	TX1	TX2	TX3	TX1	TX2	TX3
T1	5.46	5.46	5.46	0	0	0	1	1	1
T2	97.15	99.96	99.96	0	0	0	2	22	224
T3	8.81	8.81	8.81	0	0	0	1	1	1
T4	88.60	88.60	88.60	3	5	4	4	125	1445
T5	99.89	99.89	99.89	0	0	0	1	3	17
T6	94.00	99.95	99.95	0	21	21	3	153	1450
T7	7.35	7.35	7.35	1	1	1	1	1	1
T8	13.98	13.98	13.98	0	0	0	1	1	1
T9	95.28	99.97	99.97	0	0	0	4	122	1451
T10	59.96	86.79	86.79	25	26	26	3	106	1445
T11	97.68	99.94	99.94	2	2	2	6	118	1446
T12	66.27	79.16	79.16	26	26	26	3	28	267
T13	6.83	6.83	6.83	1	1	1	1	1	1
T14	60.62	60.62	60.62	3	3	3	1	1	1
T15	95.45	99.96	99.96	0	1	1	2	20	268
T16	86.40	99.86	99.86	0	0	0	1	1	4
T17	96.33	99.95	99.95	0	0	0	2	16	149
T18	99.95	99.95	99.95	2	3	3	2	53	1443
T19	97.18	97.18	97.18	11	11	11	1	4	18
T20	85.81	85.81	85.81	2	2	2	1	1	1

Table 6.8: Mythril performance results for the real-world token contract set. The coverage percentage, number of reported errors, and execution time is shown for three separate runs with a limit of 1, 2 and 3 transactions (TX).

The third and final experiment analysed tool performance and efficiency when applied to larger, more complex contracts that are currently live on the Ethereum blockchain. We analysed the top 20 ERC-20 tokens using Mythril and Manticore+ with up to three symbolic transactions. Both Mythril and Manticore+ achieved similar levels of coverage after the first transaction on successful evaluations (average of 88% vs 90%), but the latter dropped off immensely in terms of execution time.

There are several potential explanations why Manticore+ could have taken so much longer:

- Constructors were called with symbolic arguments instead of the concrete values used to deploy the live versions of these contracts;
- Enabling all the detectors has a very significant impact on execution time; and
- The underlying Manticore system forks over all possible recipients of an

ID	Coverage (%)	Errors	Time (minutes)
T1	80.72	2	14
T2	91.44	1	4
T3			
T4			
T5			
T6	94.69	0	6
T7			
T8	75.37	4	85
T9			
T10			
T11			
T12	93.92	0	30
T13			
T14			
T15	95.62	2	3
T16	87.83	0	1
T17	96.38	0	2
T18	98.47	7	2
T19			
T20			

Table 6.9: Manticore+ performance results for the real-world token contract set. The coverage percentage, number of reported errors, and execution time is shown for one transaction. Empty results failed to initialise the contract using Solidity 0.4.25.

external call or transfer of funds, causing a massive increase in paths explored. This selection of token contracts may therefore be biased against Manticore, since their primary function is to make transfers between accounts.

Chapter 7

Conclusion

We conclude by discussing the overall state of smart contract testing, the limitations and potential opportunities for future work that exist in this field, and a summary of the contributions made in this thesis.

7.1 Testing Smart Contracts

Ethereum smart contracts pose a unique security challenge for developers in that their runtime application code is always publicly visible and immutable. Without a rigorous testing and auditing process, these two characteristics can lead to immense monetary loss and disaster. One of the ways to combat this is to research and develop tools that can aid developers in automatically finding bugs.

Our work firstly focused on identifying the most common vulnerabilities in smart contracts that could be detected using security analysis tools. Although these vulnerabilities are mostly language agnostic and only found in EVM bytecode, we still opted to include known best practices and security patterns for the Solidity smart contract programming language. We ended up with 15 vulnerabilities at the Ethereum application layer that we felt were critical to our investigation.

Next, we surveyed existing security analysis tools that can automatically detect these vulnerabilities, regardless of their analysis method. We ultimately selected six tools for further discussion based on their extensibility, ease of use, and continuous integration support. One of these tools, Manticore, is a symbolic execution tool that we extended within an analysis framework named Manticore+. This system can detect a number of previously mentioned vulnerabilities that the base Manticore does not support and also synthesise adversarial smart contracts in the case of re-entrancy. The additional vulnerability detector modules and environmental setup does, however, come at the cost of longer analysis times and would make a good candidate for future work.

Finally, we evaluated these tools in terms of efficiency and effectiveness. Our first experiment tested each tool's ability to detect a known vulnerability in a set of contracts sourced from various internet repositories and articles. Securify managed to perform the best in the unauthorised Ether withdrawal and re-entrancy categories. Manticore+ performed the same as Mythril in the unauthorised self-destruct and integer overflow categories, but Mythril managed to find more assert violations.

The second experiment analysed Manticore+, Mythril, and Echidna's effectiveness at solving two online smart contract exploitation challenges: Capture the Ether and Ethernaut. Overall, the three tools managed to successfully complete 24 out of the 39 challenges, with Mythril outperforming the others. The tools (including our framework) mostly struggled in areas where they have to rely on other contracts to complete exploits, indicating that more development may be required in different approaches to synthesise attacking smart contracts.

Our final experiment evaluated our framework and Mythril's performance when analysing the top twenty ERC-20 tokens found on the main Ethereum network, so as to simulate testing a large contract before it is released. Mythril managed to successfully analyse 15 of the given contracts using our configurations and achieved more than 90% coverage in 10 of the remaining contracts after three symbolic transactions. We also observed, however, that coverage did not improve after the second transaction. Analysis times increased dramatically after each transaction, implying that the optimal configuration will likely remain at two transactions for most contracts of this scale, since many of our examples timed out after 24 hours.

The coverage results for Manticore+ was similar after the first transaction, but the system failed to analyse the majority of the given contracts in two or more transactions given our timeout parameters.

7.2 Limitations and Future Work

We only considered vulnerabilities that exist as part of the Ethereum smart contract application layer, but there is an opportunity to survey the available tooling for the rest of the Ethereum stack [34]. That study could also use new or existing tools to test the EVM clients themselves, building on the work of, for example, the EVMFuzzer team [65].

Our discussion of Ethereum vulnerabilities and tooling only focused on the Solidity language, with no consideration for other languages such as Vyper [15]. It would be beneficial to further analyse which vulnerabilities are mitigated by the Vyper compiler and determine via an empirical study if Vyper provides better security. There are, however, some concerns among developers about the maturity of Vyper and its readiness for use in production environments [87].

Other smart contract languages listed in [88] can also be considered as part of this evaluation.

The set of test contracts used as part of the vulnerability detection experiment in Section 6.2 was sourced from multiple repositories and articles, but can be further improved in several ways:

1. Seeding the corpus with samples that are guaranteed to be safe so that one can measure the rate of false-positives reported;

2. Analysing each contract (manually and tool-assisted) to determine what other vulnerabilities it may contain, since many are only labelled to contain a single vulnerability;

3. Including additional information to aid in the analysis and tool comparisons, such as the minimum transaction depth required for each bug.

By following these steps and expanding the number of contracts per vulnerability, as well as the number of different vulnerabilities featured, one would be able to more accurately compare the effectiveness of each tool.

7.3 Summary

In this work, we categorised and surveyed the most critical vulnerabilities that currently exist at the Ethereum smart contract application layer and presented best practices formulated by the research community to prevent or mitigate these exploits. A variety of security analysis tools were listed that can automatically detect vulnerabilities and extended in some cases to have better vulnerability coverage — particularly when required to synthesise adversarial smart contracts. Lastly, we evaluated these tools in terms of efficiency and effectiveness. They were able to automatically detect the majority of issues in our first two experiments, and produce on average more than 90% code coverage in 15 of the top 20 ERC-20 tokens in the case of Mythril.

Bibliography

[1] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[2] Nick Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997.

[3] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *Ethereum project white paper*, 2014.

[4] Thedao/dao-1.0: The standard dao framework at 0xbb9bc244d798123fde783fcc1c72d3bb8c189413. `https://github.com/TheDAO/DAO-1.0`. (Accessed on 12/15/2019).

[5] Osman Güçlütürk. The dao hack explained: Unfortunate take-off of smart contracts. `https://medium.com/@ogucluturk/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562`, 8 2018. (Accessed on 12/15/2019).

[6] Solidity solidity 0.5.12 documentation. `https://solidity.readthedocs.io/en/v0.5.12/`. (Accessed on 10/24/2019).

[7] Truffle Blockchain Group. Sweet tools for smart contracts — truffle suite. `https://www.trufflesuite.com/`. (Accessed on 10/29/2019).

[8] 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. *arXiv preprint arXiv:1907.03890*, 2019.

[9] Script - bitcoin wiki. `https://en.bitcoin.it/wiki/Script`. (Accessed on 10/24/2019).

[10] Roadmap — eth.wiki. `https://eth.wiki/en/roadmap`. (Accessed on 10/24/2019).

[11] Vitalik Buterin. Hard fork completed. `https://blog.ethereum.org/2016/07/20/hard-fork-completed`. (Accessed on 10/24/2019).

[12] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 2014.

[13] Ethereum mining - ethhub. `https://docs.ethhub.io/using-ethereum/mining`. (Accessed on 12/02/2019).

[14] LLL introduction LLL compiler documentation 0.1 documentation. `https://lll-docs.readthedocs.io/en/latest/lll_introduction.html`. (Accessed on 10/24/2019).

[15] Vyper Team. Vyper — vyper documentation. `https://vyper.readthedocs.io/en/v0.1.0-beta.13/`. (Accessed on 10/24/2019).

[16] Trail of Bits. Contract upgrade anti-patterns — trail of bits blog. `https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns`, 9 2018. (Accessed on 11/06/2019).

[17] Eips/eip-20.md at master ethereum/eips. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-20.md`. (Accessed on 10/24/2019).

[18] Eips/eip-721.md at master ethereum/eips. `https://github.com/ethereum/EIPs/blob/master/EIPS/eip-721.md`. (Accessed on 11/05/2019).

[19] ethereum/go-ethereum: Official go implementation of the ethereum protocol. `https://github.com/ethereum/go-ethereum`. (Accessed on 10/29/2019).

[20] ethereum/aleth: Aleth ethereum c++ client, tools and libraries. `https://github.com/ethereum/aleth`. (Accessed on 10/29/2019).

[21] ethereum/py-evm: A python implementation of the ethereum virtual machine. `https://github.com/ethereum/py-evm`. (Accessed on 10/29/2019).

[22] Ganache — ganache quickstart — documentation — truffle suite. `https://www.trufflesuite.com/docs/ganache/quickstart`. (Accessed on 10/29/2019).

[23] Ethereum for developers — ethereum. `https://www.ethereum.org/developers/#testnets-and-faucets`. (Accessed on 10/29/2019).

[24] Mocha - the fun, simple, flexible javascript test framework. `https://mochajs.org`, 10 2019. (Accessed on 11/05/2019).

[25] web3.js - ethereum javascript api web3.js 1.0.0 documentation. `https://web3js.readthedocs.io/en/v1.2.2`. (Accessed on 11/05/2019).

[26] Chai. `https://www.chaijs.com`. (Accessed on 11/05/2019).

[27] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[28] HyungSeok Han and Sang Kil Cha. Imf: Inferred model-based fuzzer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2345–2358. ACM, 2017.

[29] Michal Zalewski. american fuzzy lop. `http://lcamtuf.coredump.cx/afl`. (Accessed on 11/18/2019).

[30] Patrice Godefroid, Michael Y Levin, and David Molnar. SAGE: whitebox fuzzing for security testing. *Communications of the ACM*, 55(3):40–44, 2012.

[31] Corina S Păsăreanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *International journal on software tools for technology transfer*, 11(4):339, 2009.

[32] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *International Conference on Principles of Security and Trust*, pages 164–186. Springer, Berlin, Heidelberg, 2017.

[33] Ardit Dika. Ethereum smart contracts: Security vulnerabilities and security tools. Master's thesis, NTNU, 2017.

[34] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks and defenses. *arXiv preprint arXiv:1908.04507*, 2019.

[35] ConsenSys. Overview smart contract weakness classification and test cases. `https://swcregistry.io`. (Accessed on 10/25/2019).

[36] Safety ethereum/wiki wiki. `https://github.com/ethereum/wiki/wiki/Safety`. (Accessed on 11/20/2019).

[37] Maximilian Wohrer and Uwe Zdun. Smart contracts: security patterns in the ethereum ecosystem and solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8. IEEE, 2018.

[38] Ivica Nikolić, 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. ACM, 2018.

[39] randao/randao: Randao: A dao working as rng of ethereum. `https://github.com/randao/randao`. (Accessed on 11/15/2019).

[40] solidity - what are some examples of how inline assembly benefits smart contract development? - ethereum stack exchange. `https://ethereum.stackexchange.com/questions/3157/what-are-some-examples-of-how-inline-assembly-benefits-smart-contract-developmen`. (Accessed on 12/15/2019).

[41] Program the blockchain — signing and verifying messages in ethereum. `https://programtheblockchain.com/posts/2018/02/17/signing-and-verifying-messages-in-ethereum`. (Accessed on 12/02/2019).

[42] 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. ACM, 2018.

[43] Chainsecurity. Security scanner for ethereum smart contracts. `https://securify.chainsecurity.com`. (Accessed on 10/28/2019).

[44] Chainsecurity. Chainsecurity. `https://chainsecurity.com`. (Accessed on 10/28/2019).

[45] ETH Zurich SRI Lab. eth-sri/securify: Security scanner for ethereum smart contracts. `https://github.com/eth-sri/securify`. (Accessed on 10/28/2019).

[46] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE, 2019.

[47] Trail of bits. `https://www.trailofbits.com`. (Accessed on 10/28/2019).

[48] Trail of Bits. Crytic. `https://crytic.io`. (Accessed on 10/28/2019).

[49] Trail of Bits. crytic/slither: Static analyzer for solidity. `https://github.com/crytic/slither`. (Accessed on 10/28/2019).

[50] ConsenSys. Consensys. `https://consensys.net`. (Accessed on 10/28/2019).

[51] ConsenSys. Mythx: Smart contract security tool for ethereum. `https://mythx.io`. (Accessed on 10/28/2019).

[52] Remix - ethereum ide. `https://remix.ethereum.org`. (Accessed on 10/28/2019).

[53] Valentin Wüstholz. Checking custom correctness properties of smart contracts using mythx. `https://medium.com/consensys-diligence/checking-custom-correctness-properties-of-smart-contracts-using-mythx-25cbac5d7852`. (Accessed on 11/06/2019).

[54] Bernhard Mueller. The tech behind mythx smart contract security analysis. `https://medium.com/consensys-diligence/the-tech-behind-mythx-smart-contract-security-analysis-32c849aedaef`, 3 2019. (Accessed on 11/13/2019).

[55] Valentin Wüstholz and Maria Christakis. Learning inputs in greybox fuzzing. *arXiv preprint arXiv:1807.07875*, 2018.

[56] ConsenSys. Consensys/mythril: Security analysis tool for EVM bytecode. Supports smart contracts built for ethereum, quorum, vechain, roostock, tron and other EVM-compatible blockchains. `https://github.com/ConsenSys/mythril`. (Accessed on 10/31/2019).

[57] crytic/echidna: Ethereum fuzz testing framework. `https://github.com/crytic/echidna`. (Accessed on 10/28/2019).

[58] hevm: Ethereum virtual machine evaluator. `http://hackage.haskell.org/package/hevm`. (Accessed on 11/14/2019).

[59] State machine testing with echidna — trail of bits blog. `https://blog.trailofbits.com/2018/05/03/state-machine-testing-with-echidna`, 5 2018. (Accessed on 11/14/2019).

[60] Trail of Bits. trailofbits/manticore: Symbolic execution tool. `https://github.com/trailofbits/manticore`. (Accessed on 11/14/2019).

[61] 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. ACM, 2016.

[62] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 363–373. ACM, 2019.

[63] Johannes Krupp and Christian Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1317–1333, 2018.

[64] Bo Jiang, Ye Liu, and WK Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 259–269. ACM, 2018.

[65] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. Evmfuzzer: detect evm vulnerabilities via fuzz testing. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1110–1114. ACM, 2019.

[66] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. Smartcheck: Static analysis of ethereum smart contracts. In *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 9–16. IEEE, 2018.

[67] Patrick Ventuzelo. quoscient/octopus: Security analysis tool for webassembly module and blockchain smart contracts (btc/eth/neo/eos). `https://github.com/quoscient/octopus`. (Accessed on 11/05/2019).

[68] Protofire. Solhint - solidity linter. `https://protofire.github.io/solhint`. (Accessed on 12/05/2019).

[69] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *arXiv preprint arXiv:1809.03981*, 2018.

[70] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. *Security and Privacy*, 2020, 2019.

[71] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. Verisolid: Correct-by-design smart contracts for ethereum. *arXiv preprint arXiv:1901.01292*, 2019.

[72] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. KEVM: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.

[73] Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. Under-optimized smart contracts devour your money. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 442–446. IEEE, 2017.

[74] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):116, 2018.

[75] Yu Feng, Emina Torlak, and Rastislav Bodik. Precise attack synthesis for smart contracts. *arXiv preprint arXiv:1902.06067*, 2019.

[76] ethereum/eth-tester: Tool suite for testing ethereum applications. `https://github.com/ethereum/eth-tester`. (Accessed on 12/27/2019).

[77] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934*, 2018.

[78] Trail of Bits. crytic/etheno: Simplify ethereum security analysis and testing. `https://github.com/crytic/etheno`. (Accessed on 12/25/2019).

[79] Trail of Bits. crytic/not-so-smart-contracts: Examples of solidity security issues. `https://github.com/crytic/not-so-smart-contracts`. (Accessed on 12/11/2019).

[80] OpenZeppelin. Ethernaut. `https://ethernaut.openzeppelin.com`. (Accessed on 09/23/2019).

[81] SMARX. Capture the ether - the game of ethereum smart contract security. `https://capturetheether.com`. (Accessed on 09/23/2019).

[82] Andreas M Antonopoulos and Gavin Wood. *Mastering ethereum: building smart contracts and dapps*. O'Reilly Media, 2018.

[83] Security alert — parity technologies. `https://www.parity.io/security-alert-2`. (Accessed on 12/15/2019).

[84] OpenZeppelin. Openzeppelin/ethernaut: Web3/solidity based wargame. `https://github.com/OpenZeppelin/ethernaut`. (Accessed on 09/23/2019).

[85] Enigmatic. Smart contract exploits part 1 featuring capture the ether (lotteries). `https://medium.com/coinmonks/smart-contract-exploits-part-1-featuring-capture-the-ether-lotteries-8a061ad491b`, 9 2018. (Accessed on 09/26/2019).

[86] Etherscan. Ethereum (eth) blockchain explorer. `https://etherscan.io`. (Accessed on 09/23/2019).

[87] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach D Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. Smart contract development: Challenges and opportunities. *IEEE Transactions on Software Engineering*, 2019.

[88] Dominik Harz and William Knottenbelt. Towards safer smart contracts: A survey of languages and verification methods. *arXiv preprint arXiv:1809.09805*, 2018.

Appendix A

Attacking Contract

```
pragma solidity ^0.4.25;

contract AttackingContract {
    address vulnerableContract;
    uint counter;
    uint txCounter;

    bytes[] symbolicData;
    uint[] symbolicValues;
    bytes[] proxyData;
    bytes32[] returnData;

    constructor() payable {}

    function proxy() public payable {
        vulnerableContract.call.value(msg.value)(proxyData[txCounter]);
        txCounter += 1;
    }

    function setTarget(address _vulnerableContract, uint _counter) public {
        vulnerableContract = _vulnerableContract;
        counter = _counter;
    }

    function setProxyData(bytes data) public {
        proxyData.push(data);
    }

    function setSymbolicData(bytes data) public {
        symbolicData.push(data);
    }

```

```
33      function setSymbolicValue(uint value) public {
34          symbolicValues.push(value);
35      }
36
37      function setReturnData(bytes32 data) public {
38          returnData.push(data);
39      }
40
41      function () payable {
42          if (msg.data.length > 0) {
43              bytes32 data = returnData[txCounter];
44              assembly {
45                  let r := data
46                  mstore(0x0, r)
47                  return(0x0, 32)
48              }
49          } else if (counter > 0) {
50              counter -= 1;
51              vulnerableContract.call.value(symbolicValues[counter])(
52                  symbolicData[counter]
53              );
54          }
55      }
56  }
```

Glossary

ABI An Application Binary Interface (ABI) specifies the calling convention and input data structure formats of a binary program.

AST An abstract syntax tree (AST) represents the abstract syntax of a program in a tree structure.

Bitcoin An open-source, decentralised cryptocurrency based on a public blockchain. Launched in 2009, Bitcoin is the world's most widely traded cryptocurrency (by volume) and inspired hundreds of new cryptocurrencies since then.

blockchain An immutable, decentralised, distributed ledger represented in a tree structure. Typically they employ some form of cryptographic consensus protocol to maintain a trustless interaction between its participants and a single source of truth.

CFG A control-flow graph represents the potential paths travelled within a program using a graph. Nodes represent blocks of code and outward directed edges represent the jump destinations following these blocks.

cryptocurrency A digital asset that can typically be exchanged using cryptographic authorisation and a decentralised trustless authority like a blockchain. These currencies mainly differ from that of a traditional banking institution in that: newly issues currency are not minted by a sovereign central authority, participation in the system does not require physical proof of identity, and transaction rules are determined by source code instead of legislature and contractual agreements.

DSL Domain-specific languages are programming languages designed for use in a particular domain or application as opposed to a general-purpose language.

Ether The cryptocurrency available within Ethereum that can be freely traded between user accounts and smart contracts.

Ethereum An open-source, decentralised, distributed computing platform hosted on a public blockchain. This system also features a tradable cryptocurrency called Ether. Ether is used to pay for computation on the EVM.

EVM The Ethereum Virtual Machine (EVM) is a stack-based virtual machine that forms part of the Ethereum computational platform and is responsible for executing all hosted smart contract code.

ICO An Initial Coin Offering (ICO) is a round of funding for a new venture that utilises a cryptocurrency to raise money and manage share distribution in the investment.

IR An intermediate representation (IR) is an internal transformation of high-level source code that captures all necessary information generally required for analysis, optimisation, or further code generation. Compilers and static analysis tools will typically transform the given source code to an IR at some stage as part of their process.

smart contract The autonomous applications that are hosted on the Ethereum platform. They may possess its own Ether balance, persist data between transactions and call other smart contracts.

Solidity Object-oriented smart contract language that targets the EVM.

SSA A program is in a static single assignment (SSA) form when all variables are only assigned values once throughout their context.

Acronyms

ABI Application Binary Interface.

AST Abstract Syntax Tree.

CFG Control-flow Graph.

DSL Domain-specific Language.

EIP Ethereum Improvement Proposal.

ERC Ethereum Request for Comment.

EVM Ethereum Virtual Machine.

ICO Initial Coin Offering.

IR Intermediate Representation.

SSA Static Single Assignment.