Smart Contracts Security

Exploits in Ethereum Smart Contracts

George Giamouridis

Department of Digital Systems
University of Piraeus

This dissertation is submitted for the degree of
Bachelor of Science

April 2020

I would like to dedicate this thesis to my father and my uncle Daniel.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

George Giamouridis
April 2020

Acknowledgements

I would like to express immeasurable appreciation and deepest gratitude for the help and support are extended to the following persons who in one way or another have contributed in making this thesis possible.

There are no proper words to convey my deep gratitude and respect for my thesis and research advisor, Professor Chiristos Xenakis. He has inspired me to become an independent researcher and helped me realize the power of critical reasoning. He also demonstrated what a brilliant and hard-working scientist can accomplish.

I deeply thank my parents, Kostandinos and Stavroula for their unconditional trust, timely encouragement, and endless patience. It was their love that raised me up again when I got.

Finally, I cannot forget friends who went through hard times together, cheered me on, and celebrated each accomplishment.

Abstract

In the area of cybersecurity, no system can be considered fully secure. What is considered safe today, will not be tomorrow, given the nature of cybercrime and the intent of the criminal who is looking for new methods of attack. Although, some of the main abilities of blockchain, offer confidentiality, integrity and data availability, it is very important to embrace security control and optimal practices, for the purpose of the protection by external or internal attacks.

Cyberattacks become more and more targeted and complex, because of the advanced software. Attackers attempt to steal valuable data such as copyrights, sensitive personal data, medical documents, financial data, by performing high profitability strategies.

Blockchain is a technology which ensure to avert "cheating activities" by using consensus mechanisms so it is possible to spot data breaches. However, blockchain features cannot promise totally security.

Ethereum is one of the most popular (the second one after Bitcoin) decentralized platform which is founded by Vitalik Buterin in July 30, 2015. Ethereum is used for ruining Smart Contracts. Smart Contracts are programs that run directly on the ledger, and have become a key feature of blockchains today. This type of programs can be used to facilitate, verify, or enforce rules between the parties, allowing for direct editing, and interactions with other smart contracts. A software like that, is very vulnerable and it is very easy to attract lot of attackers.

Ethereum brings new methods for software development and therefore development standards and practices need to be implemented (and updated) to take into account the Smart Contracts life cycle. In this thesis we are going to present some of the most popular attacks on Smart Contract, how they can cause problems and how we can prevent them. Also we will suggest some general and some specific strategies and techniques in order to protect our Smart Contracts.

Table of contents

List of figures

Listings

Chapter 1

Blockchain Components

1.1 Peer to Peer (P2P) Network

In a P2P architecture, there is very little or no dependence in exclusive servers located in data centers. Conversely, application takes advantage of the direct connection between computers called *peers*. Peers do not belong in any service provider, but they are user-controlled computers. Because of the connection without any service provider, the architecture is called *peer architecture*. All network nodes interconnected in a mesh network with a flat topology. Nodes in a P2P network, both provide and consume services at the same time with reciprocity acting as the incentive of participation. This is a key feature of P2P architecture and it is known as *self-scalability*. P2P networks are inherently resilient, decentralized and open.

Blockchain networks architecture is much more than a topology choice. Blockchain networks are P2P data handling systems, and the network architecture is both a reflection and a foundation of that core characteristic. Decentralization of control, is a core design principle that can only be archived and maintained by a flat decentralized P2P consensus network. The term "Blockchain network" refers to a collection of nodes running the P2P protocol.

Plenty of toady's popular applications use P2P architecture for data sharing (BitTorrent) [9], faster downloading (Xunlei) [14] and video-conference. Note that some applications combine both C2S (Client to Server) and P2P architecture. P2P architecture is also known for its low cost, because it does not require any significant infrastructure and bandwidth. However, P2P applications face security, performance and reliability issues, because of their completely decentralized structure.

1.2 Consensus Rules

Blockchain is a global public ledger of all transaction, which everyone in the network accepts as the authoritative record of ownership. According to this case participants need to know who owns what without having to trust anyone. All traditional systems depend on a trust model that has a central authority providing a clearinghouse service, basically verifying and clearing all transactions. Blockchain networks have not central authorities to rely on. However, every node has a complete copy of the public ledger that it can trust as the authoritative record. The Blockchain does not constructed by any central authority, but is assembled independently by every node in the network. Every node in the network, acting on information transmitted across an insecure network, where connections can arrive at the same conclusion and assemble a copy of the same public ledger as everyone else.

Blockchain's main invention is the decentralized mechanism for *emergent* consensus. Emergent, because is not achieved explicitly and there is no election or fixed moment when consensus occurs. Instead, consensus is an emergent artifact of the asynchronous interaction of thousands of independent nodes, all following simple rules. All the properties of a Blockchain network, including currency, transactions, payments, and the security model that does not depend on central authority or trust, drive from this invention.

1.3 Chain of Blocks (Blockchain)

The Blockchain data structure is an ordered back-linked list of blocks of transactions. Every node in a Blockchain network stores the Blockchain metadata. Blocks are linked back, each referring to the previous block in the chain. Blocks stacked on top of each other, results in the use of terms like *height*, to refer to the distance from the first block, and *top* to refer to the most recently added block.

Each block within the Blockchain is identified by a hash, generated using a cryptographic hash algorithm on the header of the block. Each block also refers to a previous block, known as the *parent block*, through the previous block hash field in the block header. In other words, each block contains the hash of its parent inside its own header. The sequence of hashes linking each block to its parent, creates a chain going back all the way to the first block ever created, known as the *genesis block*.

While a block has just one parent, it can temporarily have multiple children. Each of the children refers to the same block as its parent and contains the same (parent) hash. Multiple children arise during a Blockchain fork, a temporary situation that occurs when different blocks are discovered almost simultaneously by different miners. Eventually, only one child

block becomes part of the Blockchain and the fork is resolved. Even though, a block may have more than one child, each block can have only one parent. This is because a block has one single previous block hash field referencing its single parent.

The "previous block hash" field is inside the block header and thereby affects the current block's hash. The child's own identity changes if the parent's identity changes. When the parent is modified in any way, the parent's hash changes. The parent's changed hash necessitates a change in the previous block hash pointer of the child. This, in turn causes the child's hash to change, which requires a change in the pointer of the grand-child, which in turn changes the grandchild and so on. This cascade effect ensures that once a block has many generations following it, it cannot be changed without forcing a recalculation of all subsequent blocks.

1.4 Consensus Algorithm

A consensus algorithm is a procedure through which all peers that participate in a Blockchain network, reach a common agreement about the state of the public ledger, where all transactions are listed. Consensus algorithms achieve reliability in the Blockchain network and establish trust between unknown peers. Essentially, the consensus protocol is used to ensure that every new block that is added to the Blockchain is valid, meaning that it is the one and only version of the truth that is agreed upon by all nodes inside the network.

Some of the most popular consensus algorithms are shown below:

- **Proof of Work (PoW):** The central idea behind this algorithm is to solve a complex mathematical problem and easily give out a solution
- **Proof of Stake (PoS):** In this type of consensus algorithm, instead of investing in expensive hardware to solve a complex problem, validators invest in the coins of the system by locking up some of their coins as stake
- **Proof of Burn (PoB):** With PoB, instead of investing into expensive hardware equipment, validators "burn" coins by sending them to an address from where they are irretrievable
- **Proof of Capacity:** In the Proof of Capacity consensus, validators are supposed to invest their hard drive space instead of investing in expensive hardware or burning coins
- **Proof of Elapsed Time:** PoET is one of the fairest consensus algorithms which chooses the next miner using fair means only

Chapter 2

Ethereum Components

2.1 Transactions

Transactions are signed messages originated by an EOA (Externally Owned Account), transmitted by the Ethereum network, and recorded on the Ethereum Blockchain. This basic definition conceals a lot of surprising and fascinating details. Another way to look at transactions, is that they are the only things that can trigger a change of state, or cause a contract to execute in the EVM (Ethereum Virtual Machine). Ethereum is a global singleton state machine, and transactions are what make that state machine changing its state. Contracts do not run on their own. Ethereum does not run autonomously. Everything starts with a transaction.

A transaction is a serialized binary message that contains the following data:

- **Nonce:** A sequence number, issued by the originating EOA, used to prevent message replay
- **Gas price:** The price of gas (in wei) the originator is willing to pay
- **Gas limit:** The maximum amount of gas the originator is willing to buy for this transaction
- **Recipient:** The destination Ethereum address
- **Value:** The amount of ether to send to the destination
- **Data:** The variable-length binary data payload
- **v, r, s:** The three components of an ECDSA (Elliptic Curve Digital Signature Algorithm) of the originating EOA

2.2 Ethereum Virtual Machine (EVM)

The EVM is the part of Ethereum that handles smart contract deployment and execution. Simple value transfer transactions from one EOA to another do not need to involve it practically speaking, but everything else will involve a state update computed by the EVM. At a high level, the EVM running on the Ethereum Blockchain can be thought of as a global decentralized computer containing millions of executable objects, each with its own permanent data store.

The EVM is a "quasi–Turing-complete state machine". Quasi, because all execution processes are limited to a finite number of computational steps by the amount of gas available for any given smart contract execution. As such, the halting problem is solved and the situation where execution might run forever, thus bringing the Ethereum platform to halt in its entirety, is avoided.

The EVM has a stack-based architecture, storing all in-memory values on a stack. It works with a word size of 256 bits and has several addressable data components:

- An immutable program code ROM, loaded with the bytecode of the smart contract to be executed
- A volatile memory, with every location explicitly initialized to zero
- A permanent storage that is part of the Ethereum state, also zero-initialized

The Halting problem

The Halting problem answers to the question "Given a program will ever halt or not?" Halting means that a program will accept a certain input and then will halt or it will reject it and then will halt. This, would never go into an infinite loop. Basically halting means terminating. Actually we cannot design a generalized algorithm which can appropriately say that given a program will ever halt or not. The only way is to run the program and check whether it halts or not.
This is an undecidable problem because we cannot have an algorithm which will tell us whether a given program will halt or not in a generalized way i.e. by having specific program/algorithm. In general, we cannot always know that is why we cannot have a general algorithm. The best possible way is to run the program and see whether it halts or not. In this way for many programs we can see that it will sometimes loop and always halt.

2.3 Consensus Algorithm

2.3.1 Ethash (Proof-of-Work Algorithm)

The current Ethereum Blockchain uses a consensus algorithm called *Ethash*. Ethash is a PoW (Proof of Work) algorithm which is actually Bitcoin's consensus model. It uses an evolution of the DAG algorithm (Dagger–Hashimoto algorithm) [23], which is a combination of Vitalik Buterin's Dagger algorithm and Thaddeus Dryja's Hashimoto algorithm.

The purpose of the DAG algorithm is to make the Ethash POW algorithm dependent on maintaining a large frequently accessed data structure. This, makes Ethash more difficult to make ASIC (Application-Specific Integrated Circuit) [22] mining equipment faster than a GPU.

Use of consumer level GPUs for carrying out the POW on the Ethereum network, means that more people around the world can participate in the mining process. The more independent miners there are, the more decentralized the mining power is, which means we can avoid a situation like in Bitcoin, where much of the mining power is concentrated in the hands of a few large industrial mining operations. The downside of GPU usage for mining is that it precipitated a worldwide shortage GPUs, causing their price to skyrocket.

Until recently, the threat of ASIC miners on the Ethereum network was largely nonexistent. Using ASICs for Ethereum, requires the design, manufacture, and distribution of highly customized hardware. Producing them requires considerable investment of time and money. The Ethereum developers long expressed plans to move to a PoS consensus algorithm likely kept ASIC suppliers away from targeting the Ethereum network for a long time. As soon as Ethereum moves to PoS, ASICs designed for the PoW algorithm will be rendered useless, unless miners can use them to mine other cryptocurrencies instead. The latter possibility is now a reality with a range of other Ethash-based consensus coins available has pledged to remain a PoW Blockchain for the foreseeable future. This means that we will likely see ASIC mining begin to become a force on the Ethereum network while it is still operating on PoW consensus.

2.3.2 Casper (Proof-of-Stake Algorithm)

Casper is the proposed name for Ethereum's PoS consensus algorithm. It is still under active research and development and it is not implemented on the Ethereum Blockchain. Casper is being developed in two competing "flavors":

- **Casper FFG:** The Friendly Finality Gadget
- **Casper CBC:** The Friendly GHOST/Correct-by-Construction

Initially, Casper FFG was proposed as a hybrid PoW/PoS algorithm to be implemented as a transition to a more permanent pure PoS algorithm. But in June 2018, Vitalik Buterin, who was leading the research work on Casper FFG, decided to scrap the hybrid model in favor of a pure PoS algorithm. Now, Casper FFG and Casper CBC are both being developed in parallel.

2.4 Clients

An Ethereum client is a software application that implements the Ethereum specifications and communicates over the P2P network with other Ethereum clients. While there are different types of clients [10], that means that they are also implemented by different teams and in different programming languages, but they all speak the same protocol and follow the same rules. As such, they can all be used to operate and interact with the same Ethereum network.

2.4.1 Full Node

The health, resilience, and censorship resistance of Blockchain, depends on them having many independently operated and geographically dispersed full nodes. Each full node can help other new nodes obtain the block data, to bootstrap their operation, as well as offering the operator an authoritative and independent verification of all transactions and contracts. However, running a full node will incur a cost in hardware resources and bandwidth.

2.4.2 Remote Client

There is also the option of running a remote client, which does not store a local copy of the Blockchain or validate blocks and transactions. These clients offer the functionality of a wallet and can create and broadcast transactions. Remote clients can be used to connect to existing networks. The terms "remote client" and "wallet" are used interchangeably,

though, there are some differences. Usually, a remote client offers an API in addition to the transaction functionality of a wallet.

Chapter 3

Smart Contracts

3.1 Smart Contract Definition

The term smart contract is defined as "a set of promises, specified in digital form, including protocols within which the parties perform on the other promises". The concept of smart contracts has evolved especially after the introduction of decentralized Blockchain platforms with the invention of Bitcoin in 2009. In the context of Ethereum, the term is actually a bit of a misnomer, given that Ethereum smart contracts are neither smart nor legal contracts, but the term has stuck. Here, we use the term smart contracts to refer to immutable computer programs that run deterministically in the context of an EVM as part of the Ethereum network protocol.

Let's explain in more details the properties of a Smart Contract:

- **Computer programs:** Smart contracts are simply computer programs. The word "contract" has no legal meaning in this context
- **Immutable:** Once deployed, the code of a smart contract cannot change. Unlike with traditional software, the only way to modify a smart contract is to deploy a new instance
- **Deterministic:** The outcome of the execution of a smart contract is the same for everyone who runs it, given the context of the transaction that initiated its execution and the state of the Ethereum Blockchain at the moment of execution
- **EVM context:** Smart contracts operate with a very limited execution context. They can access their own state, the context of the transaction that called them, and some information about the most recent blocks

- **Decentralized world computer:** The EVM runs as a local instance on every Ethereum node, but because all instances of the EVM operate on the same initial state and produce the same final state, the system operates as a single world computer

3.2 Smart Contract Life Cycle

Smart contracts are typically written in a high-level language, such as Solidity. But in order to run they must be compiled to the low-level bytecode that runs in the EVM. Once compiled they are deployed on the Ethereum platform using a special contract creation transaction, which is identified as such by being sent to the special contract creation address, namely *0x0*. Each contract is identified by an Ethereum address which is derived from the contract creation transaction as a function of the originating account and nonce.

Importantly contracts only run if they are called by a transaction. All smart contracts in Ethereum are executed ultimately, because of a transaction initiated from an EOA. A contract can call another contract that can call another contract and so on, but the first contract in such a chain of execution will have always been called by a transaction from an EOA. Contracts never run on their own or in the background. Contracts effectively lie dormant until a transaction triggers execution, either directly or indirectly as part of a chain of contract calls. It is also worth noting that smart contracts are not executed in parallel in any sense. The Ethereum world computer can be considered to be a single-threaded machine.

Transactions are atomic, regardless of how many contracts they call or what those contracts do when called. Transactions execute in their entirety, with any changes in the global state recorded only if all execution terminates successfully. Successful termination means that the program executed without any error and reached the end of execution. If execution fails due to an error, all of its effects are rolled back as if the transaction never ran. A failed transaction is still recorded as having been attempted, and the ether spent on gas for the execution is deducted from the originating account, but it otherwise has no other effects on contract or account state.

It is important to remember that a contract's code cannot be changed. However, a contract can be deleted, removing the code and its internal state from its address leaving a blank account. Any transactions sent to that account address after the contract has been deleted, do not result in any code execution, because there is no longer any code there to execute.

3.3 Ethereum High-Level Languages

The EVM is a virtual machine that runs a special form of code called *EVM bytecode*, analogous to every computer's CPU.

While it is possible to program smart contracts directly in bytecode, EVM bytecode is rather unwieldy and very difficult for programmers to read and understand. Instead, most Ethereum developers use a high-level language to write programs, and a compiler to convert them into bytecode.

In general, programming languages can be classified into two broad programming paradigms: *declarative* and *imperative*, also known as *functional* and *procedural*, respectively.

- **Declarative Programming:** In declarative programming we write functions that express the logic of a program, but not its flow. Declarative programming is used to create programs where there are no side effects, meaning that there are no changes to state outside a function.
- **Imperative Programming:** Imperative programming is where a programmer writes a set of procedures that combine the logic and flow of a program.

While imperative programming is more commonly used by programmers, it can be very difficult to write programs that execute exactly as expected. The ability of any part of the program to change the state of any other, makes it difficult to reason about a program's execution and introduces many opportunities for bugs. Declarative programming by comparison, makes it easier to understand how a program will behave. Since it has no side effects, any part of a program can be understood in isolation.

In smart contracts, bugs literally cost money. As a result, it is critically important to write smart contracts without unintended effects. So, declarative languages play a much bigger role in smart contracts than they do in general-purpose software. Nevertheless, the most widely used language for smart contracts (Solidity) is imperative.

Currently, supported high-level programming languages for smart contracts include:

- LLL
- Serpent
- Solidity
- Vyper
- Bamboo

3.4 The ABI Contract

In computer software, an ABI (Application Binary Interface) is an interface between two program modules. Often between the operating system and user programs. An ABI defines how data structures and functions are accessed in machine code. This is not to be confused with an API which defines this access in high-level, often human-readable formats as source code. The ABI is thus the primary way of encoding and decoding data in and out of machine code.

In Ethereum, the ABI is used to encode contract calls for the EVM and to read data out of transactions. The purpose of an ABI is to define the functions in the contract that can be invoked and describe how each function will accept arguments and return its result.

A contract's ABI is specified as a JSON array of function descriptions and events. A function description is a JSON object with fields *type*, *name*, *inputs*, *outputs*, *constant*, and *payable*. An event description object has fields *type*, *name*, *inputs*, and *anonymous*.

3.5 Gas Consideration

Gas is the fuel of Ethereum. Gas is not ether. It is a separate virtual currency with its own exchange rate against ether. Ethereum uses gas to control the amount of resources that a transaction can use, since it will be processed on thousands of computers around the world. The open-ended computation model requires some form of metering in order to avoid DOS attacks or inadvertently resource-devouring transactions.

Gas is separate from ether in order to protect the system from the volatility that might arise along with rapid changes in the value of ether, and also as a way to manage the important and sensitive ratios between the costs of the various resources that gas pays for.

3.5.1 The gasPrice Field

The *gasPrice* field in a transaction allows the transaction originator to set the price they are willing to pay in exchange for gas. The price is measured in wei per gas unit. Wallets can adjust the *gasPrice* in transactions they originate to achieve faster confirmation of transactions. The higher the *gasPrice*, the faster the transaction is likely to be confirmed. Conversely, lower priority transactions can carry a reduced price, resulting in slower confirmation. The minimum value that *gasPrice* can be set to is zero, which means a fee free transaction.

3.5.2 The gasLimit Field

The *gasLimit* field gives the maximum number of units of gas the transaction originator is willing to buy in order to complete the transaction. For simple payments, meaning transactions that transfer ether from one EOA to another EOA, the gas amount needed is fixed at 21,000 gas units.

As we mention before, gas is a resource constraining the maximum amount of computation that Ethereum will allow a transaction to consume. If the gas limit is exceeded during computation, the following series of events occurs:

- An *out of gas* exception is thrown.
- The state of the contract prior to execution is restored.
- All ether used to pay for the gas is taken as a transaction fee. It is not refunded.

Because gas is paid by the user who initiates the transaction, users are discouraged from calling functions that have a high gas cost. It is thus in the programmer's best interest to minimize the gas cost of a contract's functions. To this end, there are certain practices that are recommended when constructing smart contracts, to minimize the gas cost of a function call.

Chapter 4

Smart Contracts Security

Security is one of the most important considerations when writing smart contracts. In the field of smart contract programming, mistakes are costly and easily exploited. As with other programs, a smart contract will execute exactly what is written, which is not always what the programmer intended. Furthermore, all smart contracts are public, and any user can interact with them simply by creating a transaction. Any vulnerability can be exploited, and losses are almost always impossible to recover. It is therefore critical to follow best practices and use well-tested design patterns.

4.1 General Secure Coding Practices

- **Simplicity:** Complexity is the enemy of security. The simpler the code, and the less it does, the lower the chances are of a bug or unforeseen effect occurring. When first engaging in smart contract programming, developers are often tempted to try to write a lot of code. Instead, we should look through our smart contract code and try to find ways to do less, with fewer lines of code, less complexity, and fewer features.
- **Code reuse:** If a library or contract already exists that does most of what we need, reuse it. If we see any snippet of code repeated more than once, we should think if it could be written as a function or library and reused. Code that has been extensively used and tested is likely more secure than any new code we write.
- **Code quality:** Writing DApps in Solidity is not like creating a web widget in JavaScript. Rather, we should apply rigorous engineering and software development methodologies.

- **Readability:** The code should be clear and easy to comprehend. The easier it is to read, the easier it is to audit. Smart contracts are public, as everyone can read the bytecode and anyone can reverse engineer it. Therefore, it is beneficial to develop our work in public, using collaborative and open source methodologies. We should write code that is well documented and easy to read, following the style and naming conventions that are part of the Ethereum community.
- **Test coverage:** Smart contracts run in a public execution environment, where anyone can execute them with whatever input they want. We should test all arguments to make sure they are within expected ranges and properly formatted before allowing execution of our code to continue.

4.2 Smart Contract Attacks

In this section, we are going to demonstrate some of the most popular attacks against Smart Contracts. We will explain how each exploit works and what problems can cause, and after that we will suggest some preventing techniques to avoid those attacks.

4.2.1 Double Spending

Double Spending attack is a two-step attack which is not occurred only in the Ethereum Blockchain, but in most Blockchain networks. The attacker aims to revert a transaction which has already created and deploy.

The Attack

Stealth mining

Stealth mining is the first step of the attack. When a block is mined, it is deployed to the network in order to be validated by all other miners. This process does not always happen. In order to attempt a stealth mining attack, the attacker must create a private clone of the entire Blockchain. When miners try to add blocks in this chain, the attacker try on his own to mine every block but in the end he does not deploy it on the network.

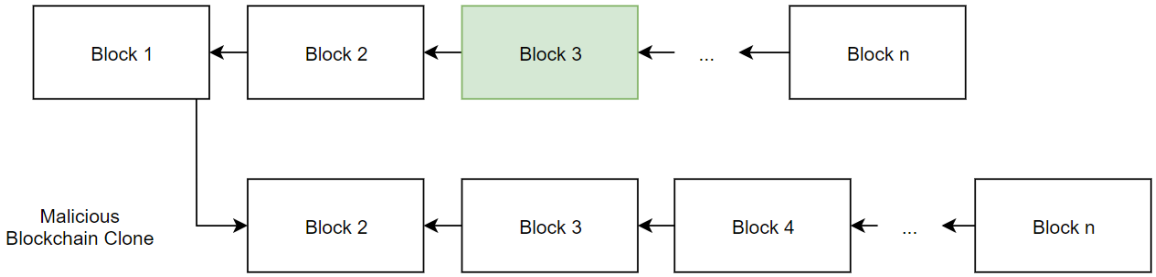

Fig. 4.1 Stealth Mining in Double Spending

Block 3 (colored green), is the block which contains the malicious transaction. Conversely, block 3 in the *Malicious Blockchain Clone* does not contain the malicious contract.

The attacker receives blocks from the Blockchain and validates them alone. In this case if the attacker try to publish his own chain, the network will reject it because the original chain is the longer one. Attacker's chain is not longer than the original chain, so the publication will be fail.

51% Attack

One of the most popular Blockchain attacks is the *51% attack*. Technology behind Blockchain based on democratic administration, meaning that decisions are taken from the majority. Thus, if most of the users follow another chain (which contains other transactions), different from the original, the transactions of the original chain will be canceled and all the ether which have spent until this time will be return to their owners.

In Figure 4.2 all participants follow the *green* chain which is the original one. In order for the attacker to perform a 51% attack, must have more processing power from the remaining network or to find a vulnerability on the block constructing protocol and speed up the process of adding the block in his own chain.

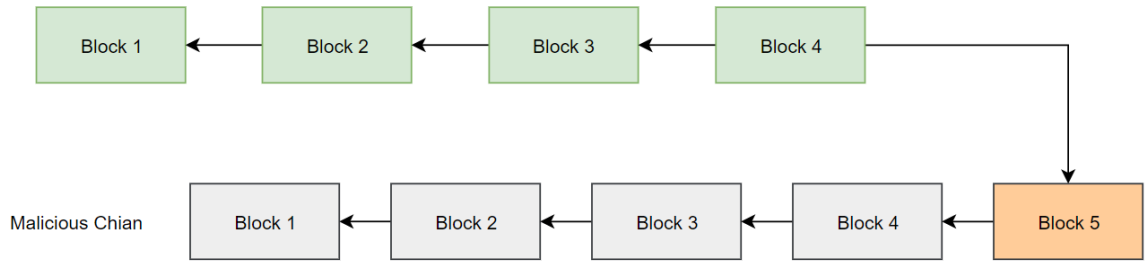

Fig. 4.2 Constructing the malicious chain in 51% attack

After the attacker successfully perform the attack, his private chain is longer than the original public chain, so now can publish it without any problem. That means that all the other participants will now follow this chain and the balance of their wallets will be updated. The malicious chain is now a *trust chain* and all transactions that are not existed in it, will be overridden directly. Now the network will follow the *orange* chain as it showed in Figure 4.3.

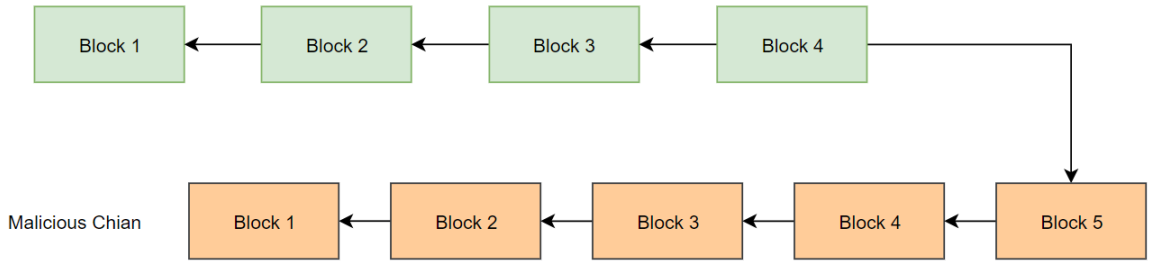

Fig. 4.3 Publication of the malicious chain

Preventative Technique

This type of attack is being prevented by the real cost of the attack and the properties of the Blockchain platform. Also, token price in the global market is a very important factor. The higher the value of the token, the more precise the attack becomes. Finally, an attack which can affect the entire Blockchain network, can be restarted in a specific time and return in the state which was before the attack.

4.2.2 Reentrancy Attack

Reentrancy Attack is an exploit which is based on external contract calling meaning that an external malicious contract can take over the control flow and make changes to data. Contracts typically handle ether and as such often send ether to various external user addresses. These operations require the contracts to submit external calls. These external calls can be hijacked by attackers who can force the contracts to execute further code (through the fallback function), including calls back into themselves. This type of attack can occur when a smart contract sends ether to an unknown address. A malicious user can construct a contract at an external address that contain malicious code in the fallback function. Thus, when a contract send ether to this address the vulnerable code will be executed.

The Attack

Consider the two contracts in Figure 4.4 where contract *B* is the malicious contract. The contract *A* sends some ether to contract *B*. The contract *A* will do an external call to contract *B* and contract *B* will execute the fallback function to handle those ether that contract *A* has sent. The malicious code exists in the fallback function, which means that when the fallback function will be executed, it will force contract *A* to perform some operations that are not expected to happen and that they obviously will cause confusion.

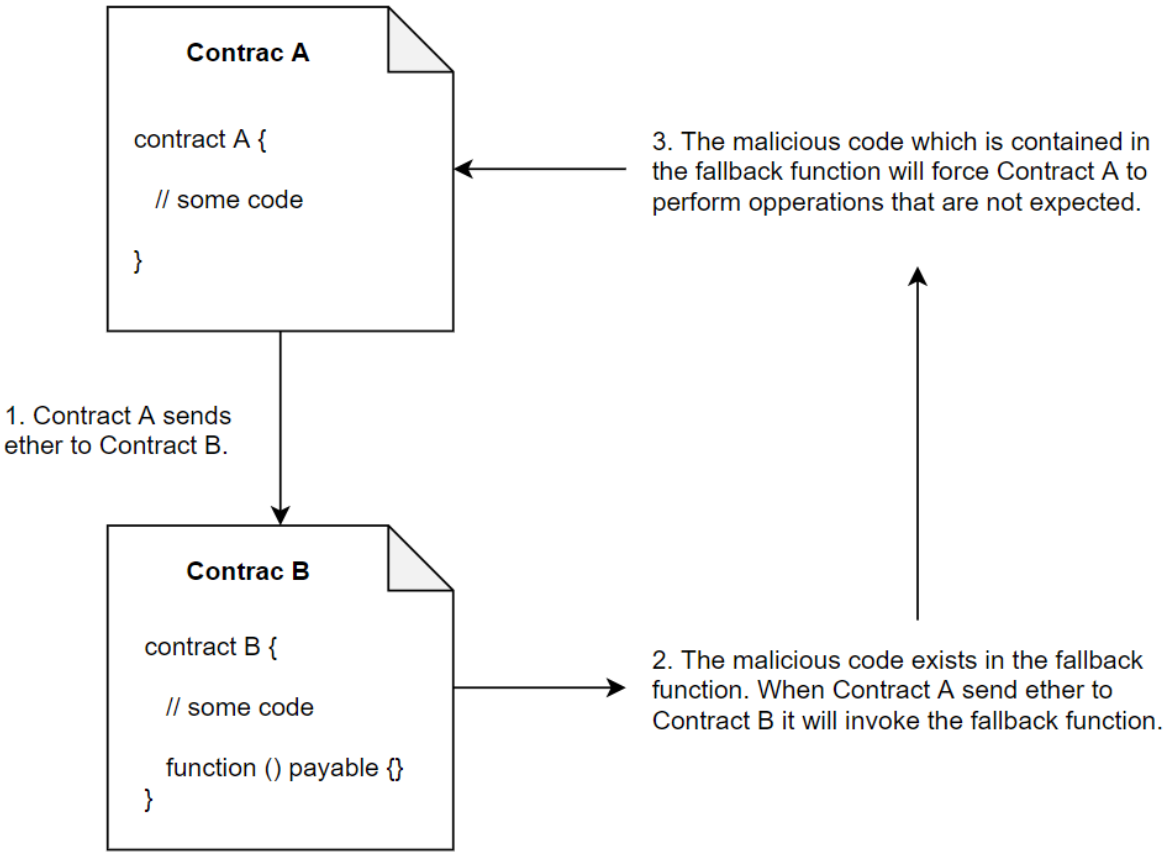

Fig. 4.4 Reentrancy Attack workflow

Consider now we are having the contract *Etherstore* with a balance of 5 ether and a malicious contract *Attack* with a zero balance.

```
contract EtherStore {

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

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

Listing 4.1 EtherStore contract

```
import "EtherStore.sol";

contract Attack {

  EtherStore public etherStore;

  constructor(address _etherStoreAddress) {
      etherStore = EtherStore(_etherStoreAddress);
  }

  function attackEtherStore() external payable {
      etherStore.$deposit$Funds.value(1 ether)();
      etherStore.withdrawFunds(1 ether);
  }

  function () payable {
      if (etherStore.balance > 1 ether) {
          etherStore.withdrawFunds(1 ether);
      }
  }
}
```

Listing 4.2 Attack contract

The contract Etherstore consist of two functions. The function *deposit* which just increments sender's balance and the *withdrawFund* function, which allows sender to specify an amount of ether to withdraw and also checks that the sender has enough balance and that the requested amount is less than 1 ether. If those two requirements are not true the function will fail. Let's see now, what is going to happen when the contract Etherstore try to send some ether to Attack contract.

In **line 12** the Attack contract will call the *deposit* function of contract Etherstore. The function will be called and executed with a parameter of 1 ether, so now the contract Attack has a balance of 1 ether.

After that, in **line 13** the Attack contract will call the *withdrawFund* function of contract Etherstore with a parameter of 1 ether. As the balance of the Attack contract is 1 ether (because of line 12 and also the amount of withdraw is 1 ether because of the parameter we have passed in line 13), the requirements in **lines 8 - 9** of EtherStore contract will pass successfully.

Then, in **line 10** the Etherstore contract will send 1 ether to Attack contract, so now the fallback function will be executed. The fallback function is executed when the contract has to handle an amount of ether.

In the beginning, the total balance of EtherStore contract was 5 ether, so now after the execution of the *withdrawFund* function is 4 ether. That means that now in **line 16** of Attack contract where the fallback function is called, the condition in **line 17** will pass successfully. Thus, the fallback function calls the EtherStore *withdrawFund* function again in **line 18** and *reenters* the EtherStore contract.

In this second call of the *withdrawFund* function, the Attack contract balance is still 1 ether because **line 11** of EtherStore contract has not executed yet. So, all the requirements (**lines 8 - 9**) are passed. Now the Attack contract withdraw 1 more ether.

This process will repeat until it is no longer the case that *EtherStore.balance > 1 ether*, meaning that EtherStore contract left with a balance of less than or equal to 1 ether.

Once there is 1 or less ether left in the EtherStore contract, this "if" statement in **line 17** inside the fallback function will fail. This, will then allow **line 11** of EtherStore contract to be executed. The balance in line 11 of EtherStore contract will be set and the execution will terminate.

The result is that the attacker has withdrawn all but 1 ether from the EtherStore contract in a single transaction.

As we can see, what is happening here, is the execution of a recursive loop which actually forces contract EtherStore to execute the *withdrawFund* function until the contract left with a minimum or zero balance.

Preventative Technique

In order to prevent this problematic situation, we just have to make a minor change to our code. The only thing we have to change is the position of **line 11** in EtherStore contract. We will move this line before **line 10**. Now, the new code of EtherStore contract (Prevent contract) is presented in Listing 4.3 (while the Attack contract remaining the same).

When **line 12** of Attack contract executed, the *deposit* function of Prevent contract will be called and executed with a parameter of 1 ether so now contract Attack has a balance of 1 ether.

Then, in **line 13** the Attack contract will call the *withdrawFund* function with a parameter of 1 ether. **Lines 8 - 9** of Prevent contract will pass successfully.

Now, the balance of the Attack contract will be decreased in **line 10** of Prevent contract, so now contract Attack has a zero balance.

In **line 11** the fallback function will be invoked and executed successfully so the payment will be succeeded. The fallback function will call the *withdrawFund* again but now the balance of Attack contract is zero. Thus, **line 8** of Prevent contract will fail, so the *withdrawFund* will not executed and the payment will not succeed for a second time.

```
1  contract Prevent {
2
3      function depositFunds() external payable {
4          balances[msg.sender] += msg.value;
5      }
6
7      function withdrawFunds (uint256 _weiToWithdraw) public {
8          require(balances[msg.sender] >= _weiToWithdraw);
9          require(_weiToWithdraw <= withdrawalLimit);
10         balances[msg.sender] -= _weiToWithdraw;
11         require(msg.sender.call.value(_weiToWithdraw)());
12     }
13 }
```

Listing 4.3 Prevent contract

By using this technique we prevent the execution of this recursive and malicious loop, and we allow someone to execute a withdrawal only if the account balance is enough.

4.2.3 Arithmetic Over/Underflows

EVM defines a constant size for all data types and so for the integers. Thus, when we use a variable to store an integer, there is a specific range of numbers this variable can take. For example, a uint8 variable can store numbers between 0 and 255.

Arithmetic Overflows and Underflows

Arithmetic Overflow is a condition that occurs when an operation produces a result that is greater than a given register or storage location can store or represent.

Consider we are having a uint8 variable called *sum*. This variable can take values from 0 to 255 as the range of a uin8 variables is [0, 255]. The variable *sum* is the sum of the variable *a* (where *a* = 0) and *b* (where *b* = 257).

If we add *a* and *b* we get $sum = a + b => sum = 0 + 257 => sum = 257$. At this point notice that $sum = 257 \notin [0, 255]$. In order to store this value in the *sum* variable we have to *wrap around* the number 257 and convert it into a number which belongs in [0, 255]. For this process we use the following formula:

$$(numberToBeAdded - maximum) + minimum - 1 \tag{4.1}$$

By using 4.1 we have $sum = (257 - 255) + 0 - 1 => sum = 1 \in [0, 255]$. After 257 wrapped around, it converted into 1 which belongs in [0, 255]. Now the value 1 can be normally stored into *sum* variable.

This condition is called *arithmetic overflow*. When an overflow occurs the EVM uses the *wrap around* process so it can store the value into the variable.

We can now follow the same process when an *arithmetic underflow* occurs. Consider again we are having a uint8 variable called *sub*. This variable can take values from 0 to 255. The variable *sub* is the subtraction of the variable *a* (where *a* = 0) minus *b* (where *b* = 1).

If we subtract *b* from *a* we get $sub = a - b => sum = 0 - 1 => sub = -1$. At that point notice that $sub = -1 \notin [0, 255]$. In order to store this value in the *sub* variable we have to *wrap around* again the number -1. For this process we use the following formula:

$$maximum + (minimum - numberToBeSubtracted) + 1 \tag{4.2}$$

By using 4.2 we have $sub = 255 + (0 - 1) + 1 => sub = 255 \in [0, 255]$. After -1 wrapped around, it converted into 255 which belongs in [0, 255]. Now, the value 255 can be normally stored into *sum* variable.

The Attack

This vulnerability occurs when a mathematical equation uses a fixed size variable to store a value which is out of its range. Token contract (4.4) uses the *transfer* function to check if the user has the required balance, in order to transfer the desired amount to an address.

```
1  contract Token {
2
3    mapping(address => uint8) balances;
4    uint8 public totalSupply;
5
6    function Token(uint8 _initialSupply) {
7      balances[msg.sender] = totalSupply = _initialSupply;
8    }
9
10   function transfer(address _to, uint8 _value) public returns (bool)
       {
11     require(balances[msg.sender] - _value >= 0);
12     balances[msg.sender] -= _value;
13     balances[_to] += _value;
14     return true;
15   }
16 }
```

Listing 4.4 Token contract

Consider that the user has a zero balance account and willing to transfer 100 ether. When **line 11** inside *transfer* function executed the following will happen:

$$balances[msg.sender] - value = 0 - 100 = -100$$

Notice that we have a uint8 variable so $-100 \notin [0,255]$. So, we have an *underflow*. As we told before, the wrap around process will be executed so, value -100 will be converted into 156. Now $156 \in [0,255]$. As a result:

$$balances[msg.sender] - value = 156 \geqslant 0 = True$$

What the malicious user has achieved here, is that successfully transferred 100 ether from a zero balance account.

Preventative Technique

To avoid this type of attack we can use some specific mathematical libraries. Those libraries replace every mathematical operator with pre-made functions.

One of the most popular libraries for this reason is the *Safe Math Library* from Open Zeppelin [16]. This library replaces all the mathematical operators with functions and whenever an underflow or overflow occurs the function throws an error and the program terminates without cause any confusion. Below, is presented the Safe Math Library:

```
1  library SafeMath {
2
3      function mul(uint256 a, uint256 b) internal pure returns (uint256
           ) {
4          if (a == 0) {
5              return 0;
6          }
7          uint256 c = a * b;
8          assert(c / a == b);
9          return c;
10     }
11
12     function div(uint256 a, uint256 b) internal pure returns (uint256
           ) {
13         uint256 c = a / b;
14         return c;
15     }
16
17     function sub(uint256 a, uint256 b) internal pure returns (uint256
           ) {
18         assert(b <= a);
19         return a - b;
20     }
21
22     function add(uint256 a, uint256 b) internal pure returns (uint256
           ) {
23         uint256 c = a + b;
24         assert(c >= a);
25         return c;
26     }
27 }
```

Listing 4.5 SafeMath Library

Now, if we use Safe Math Library, the Token Contract will be the following:

```
contract Token {

  mapping(address => uint8) balances;
  uint8 public totalSupply;

  function Token(uint8 _initialSupply) {
    balances[msg.sender] = totalSupply = _initialSupply;
  }

  function transfer(address _to, uint8 _value) public returns (bool)
     {
    require(balances[msg.sender].sub(_value) >= 0);
    balances[msg.sender].sub(_value);
    balances[_to].add( _value);
    return true;
  }
}
```

Listing 4.6 contract Token with SafeMath Library

4.2.4 Default Visibilities

Solidity functions have visibility specifiers which define how these functions can be called. Each of these specifiers settle on when a function can be called either from an external user or another contract. There are four different visibility specifiers which can be used on their own or together.

- **External:** External visibility means that the function can be called by other contracts and through transactions. A function like that cannot be called internally in the same contract. Furthermore, external functions are more effective when they handle more data.
- **Public:** Public visibility means that the function can be called either inside the contract or by sending messages.
- **Internal:** Internal visibility means that the function can be used and called from the same contract or other contracts.
- **Private:** Private visibility means that the function can be called only inside the contract which has been defined.

All functions have the "public" as a default type, which allows other users to call them individually.

The Attack

This vulnerability is based on programming mentality. The default type of each function is the public. Many times, programmers do not change the public type into private type, or they do not define any type. As a result, other users can have access in private code.

The HashForEther contract in Listing 4.7 presents an address-guessing bounty game. In order to win the remaining balance of the contract, the user must generate an address whose last 8 hex characters are 0. If this happens, the HashForEther contract calls the *sendWinnings* function and sends the remaining balance of the contract to winner's address.

The problem here is that the visibility of the function is set and left in public. Thus, every other user can call it without needing to take place in the game and transfer the remaining contract balance to its own address.

```
1  contract HashForEther {
2
3      function withdrawWinnings() {
4          require(uint32(msg.sender) == 0);
5          _sendWinnings();
6      }
7
8      function _sendWinnings() {
9          msg.sender.transfer(this.balance);
10     }
11 }
```

Listing 4.7 HashForEther contract

Preventative Technique

A good practice to prevent this attack, is to always specify the visibility of the function inside a contract, even if the function is public. Recent versions of Solidity, during the compiling process warns of functions that have no explicit visibility.

4.2.5 Entropy Illusion

Ethereum Smart Contracts are deterministic state transition operations. That means that every transaction modifies the global state of the Ethereum ecosystem in a calculable way with no uncertainty. So, we can conclude that inside the Ethereum ecosystem there is no entropy or any other type of randomness. In order to achieve decentralized entropy inside a contract there must be found another way to implement it.

The Attack

Some first contracts built on Ethereum were based on *gambling*. Gambling requires randomness, and that makes the construction process of a gambling systems difficult. It is clear that the uncertainty must come from a source external to the Blockchain. This is possible for bets between players. However, it is difficult if we want to implement a contract so it can act as "third party". A common mistake is to use *feature block variables*. A feature block variable is a variable which contains information about the transaction block whose values are not yet known, such as hashes, timestamps, block numbers, or gas limits. The problem with those parameters is that there are not random, but on the contrary are depend on the miner who mines the block.

Consider we are having a contract which acts as a roulette. This contract returns a red number whenever the next block is mined and added to the Blockchain, and also its last digit is the hash of an odd number. The attacker can bet a big amount of ether if the next color is the red. If they solve the next block and find the hash ends in an odd number, they could not publish their block and mine another, until they find a solution with the block hash being an even number. Using past or present variables can be even more devastating. Furthermore, using solely block variables means that the pseudorandom number will be the same for all transactions in a block, so an attacker can multiply their wins by doing many transactions within a block.

Preventative Technique

The source of entropy must be external to the Blockchain. This can be done among peers with systems such as commit–reveal, or via changing the trust model to a group of participants. This can also be done via a centralized entity that acts as a randomness oracle. Block variables should not be used to source entropy, as they can be manipulated by miners.

4.2.6 Race Conditions

The combination of external contracts calling and multi-user nature of the underlying Blockchain, gives rise to a variety of potential Solidity pitfalls whereby users race code execution to obtain unexpected states.

The Attack

Ethereum clients pool transactions and form them into blocks. A transaction is considered valid only if it is included in a block and added in the Blockchain by a miner. Transactions are listed in the list due to their gasPrice (transactions with more gasPrice are listed on the top).

The attacker can monitor the transactions which are waiting for validation in the list until find the one who want. After that, the attacker can edit this transaction or recall its rights or change the contract's status. Next the attacker can steal all the transaction data and create another transaction with more gasPrice, so it could be listed higher in the transaction waiting lists.

Consider the contract in Listing 4.8. In this contract all participants try to find the plain text of the following hash by using the SHA3 algorithm.

```
0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44d5479b630ee0a
```

```
1  contract FindThisHash {
2      bytes32 constant public hash = 0xb5b5b97fafd9855eec9b41f74dfb6c3
3          8f5951141f9a3ecd7f44d5479b630ee0a;
4
5      constructor() public payable {}
6
7      function solve(string solution) public {
8          require(hash == sha3(solution));
9          msg.sender.transfer(1000 ether);
10     }
11 }
```

Listing 4.8 FindThisHash contract

This contract has a 500 ether balance which are going to be transferred into winner's wallet. Let's say that the plain text which gives the above hash is the string *Ethereum!*. By calling the *solve* function and using any random string as a parameter, the user can take place in the game.

The attacker is monitoring the transaction waiting list, and waiting until the transaction with the right answer founded. When the right transaction spotted, its validity is checked and the attacker construct a new transaction with the same content, with much more ether for the gasPrice. At this time, it is very possible that the miner who is going to mine the next block, to import the malicious transaction in the block, because it will be in a higher position than the original in the transaction waiting list. Finally, the attacker will receive the 500 ether from the contract.

Preventative Technique

There are two different types of attacker for this attack.

- Those who edit ether which is used for the gasPrice in the transactions
- The miners who are able to change the order of the transactions which are waiting in the transaction waiting list

A contract which belongs in the first category is much more vulnerable than a contract which belongs in the second one. That happens because the miner can only attack when a block is mined, a process which is very difficult to happen for every individual miner who aims a specific block.

To solve this problem we can set a new variable inside the contract. Its value will define the maximum ether which will be used for the gasPrice. The attacker will not be able to increase the gasPrice infinitely. As a result, the malicious transaction could not be ordered in a higher position inside the transaction waiting list. This technique is working only for the first category. Users of the second category can still attack to the contract, because they can control the order of the transactions in their own way.

Alternatively, a more reliable method for preventing this type of attack, is to use the *commit-revel system*. This type of system, let users create transactions with hidden information. When a transaction attached in a block, user send an extra transaction which includes the hidden information of the first transaction. This method is used for both categories. However, this method cannot hide the transaction value which is a very important information.

4.2.7 Denial of Service (DoS Attack)

This type of attack is very general, but includes different types of attacks that could destroy a contract's functionality. As a result, a non-fictional contract contains an inaccessible balance, so it becomes useless.

The Attack

There are several ways a contract can become non-functional.

Looping through externally manipulated mappings or arrays

This type of vulnerability occurs when the owner of the contract want to share an amount of ether with some inventors. In order to do this, the owner construct a function called *distribute* as it is shown in Listing 4.9.

```
1  contract DistributeTokens {
2      address public owner;
3      address[] investors;
4      uint[] investorTokens;
5
6      function invest() public payable {
7          investors.push(msg.sender);
8          investorTokens.push(msg.value * 5);
9      }
10
11     function distribute() public {
12         require(msg.sender == owner);
13         for(uint i = 0; i < investors.length; i++) {
14             transferToken(investors[i],investorTokens[i]);
15         }
16     }
17 }
```

Listing 4.9 DistributeTokens contract

The problem here is inside the "for" loop in **line 13**. List *investors* can be artificially inflected if the attacker create many users and then add them in the list. As a result, the **gas** which required for the execution of the "for" loop, will be much more than the contract's gas limit so the *distribute* function will not be executed.

Owner operations

Another common pattern is where owners have specific privileges in contracts and must perform specific task for the contract to proceed to the next state. In Listing 4.10 is presented a piece of code of the ICO Contract [11]. This contract requires the owner to execute the *finalize* function in order to transfer some ether.

```
...

bool public isFinalized = false;
address public owner;

    function finalize() public {
        require(msg.sender == owner);
        isFinalized == true;
    }
...

    function transfer(address _to, uint _value) returns (bool) {
        require(isFinalized);
        super.transfer(_to,_value)
    }

...
```

Listing 4.10 ICO contract

In this case, if the privileged users lose their private keys or becomes inactive, the entire contract becomes inoperable. The owner cannot call the *finalize* function anymore, so any ether cannot be transferred.

Processing state based on external calls

Contracts are sometimes send ether to an external address. Sometimes also contracts stop working until they receive an input from an external source in order to work again. Those two practices can drive into DoS attacks when the external calling fails or is prevented for external reasons. For example, when transferring ether, the attacker can construct a contract which does not accept ether transfers. If the original contract requires to withdraw some ether in order to continue the execution, the contract will never reach this state because it is unable to transfer ether to a rejected address.

Preventative Technique

In the first category, contracts should not loop through data structures that can be manipulated by external users. A withdrawal pattern is recommended, whereby each of the investors call the *withdraw* function to claim ether independently.

In the second category, a privileged user is required to change the state of the contract. In such examples, a fail safe can be used in the event that the owner becomes incapacitated. A solution is to make the owner a multisig contract [3]. Another solution is to use a time-lock. In Listing 4.10 the *require* statement on **line 13** could include a time-based mechanism, such as *require(msg.sender == owner || now > unlockTime)*, that allows any user to finalize after a period of time specified by *unlockTime*.

The previous kind of mitigation technique can be used in the third category as well. If external calls are required to progress to a new state, account for their possible failure and potentially add a timebased state progression in the event that the desired call never comes.

4.2.8 Uninitialized Storage Pointers

EVM stores data either as storage or as memory. A very common mistake is the misunderstanding of how exactly this is done and which are the default types for local variables or functions when developing contracts. As a result, it is possible to produce vulnerable contracts by inappropriately initializing variables.

The Attack

Local variables within functions, default to storage or memory depending on their type. Uninitialized local storage variables may contain the value of other storage variables in the contract. This fact can cause unintentional vulnerabilities, or be exploited deliberately.

Consider the NameRegistrar Contract on Listing 4.11 which is just a register name contract.

```
1 contract NameRegistrar {
2
3     bool public unlocked = false;
4     struct NameRecord {
5         bytes32 name;
6         address mappedAddress;
7     }
8
9     mapping(address => NameRecord) public registeredNameRecord;
```

```
10      mapping(bytes32 => address) public resolve;
11
12      function register(bytes32 _name, address _mappedAddress) public {
13          NameRecord newRecord;
14          newRecord.name = _name;
15          newRecord.mappedAddress = _mappedAddress;
16          resolve[_name] = _mappedAddress;
17          registeredNameRecord[msg.sender] = newRecord;
18          require(unlocked); // only allow registrations if contract is
                unlocked
19      }
20  }
```

Listing 4.11 NameRegistra contract

When the contract is *unlocked* every user can register a name and map it to an address. The register is initially locked, and *require* statement on **line 25** prevents register from adding name records. Now, it seems that the contract is unusable, as there is no way to unlock the registry.

Solidity Storage Mechanism

In Solidity, variables are stored in slots as they appear in the contract. Thus, *unlock* exists in **slot[0]**, *registeredName* in **slot[1]** and *resolve* in **slot[2]**. Each of these slots is 32 bytes. The boolean variable *unlocked* will look like *0x000...0* (64 0s, excluding the 0x) for false or *0x000...1* (63 0s) for true.

Solidity stores complex data types (such as structs) in storage when initializing them as local variables by default. Therefore, *newRecord* on **line 18** defaults to storage. The vulnerability is caused by the fact that *newRecord* is not initialized. Because it defaults to storage, it is mapped to storage **slot[0]**, which currently contains a pointer to unlocked. Notice that on **lines 19 and 20** we then set *newRecord.name* to *_name* and *newRecord.mappedAddress* to *_mappedAddress*. This updates the storage locations of **slot[0]** and **slot[1]**, which modifies both *unlocked* and the storage slot associated with *registeredNameRecord*.

This means that *unlocked* can be directly modified, simply by the bytes32 *_name* parameter of the *register*() function. Therefore, if the last byte of *_name* is nonzero, it will modify the last byte of storage **slot[0]** and directly change unlocked to true. Such *_name* values will cause the *require* call on **line 18** to succeed, as we have set unlocked to true. Note that the function will pass, if we use a *_name* of the form:

```
0x0000000000000000000000000000000000000000000000000000000000000001
```

Preventative Technique

The Solidity compiler shows a warning for uninitialized storage variables. Developers should pay careful attention to these warnings when building smart contracts. The current version of Mist [15] does not allow these contracts to be compiled. It is often good practice to explicitly use the *memory* or storage specifiers when dealing with complex types, to ensure they behave as expected.

4.2.9 Block Timestamp Manipulation

Block timestamps have been used for a variety of applications, such as entropy for random numbers as we explained in before, locking funds for periods of time, and various state-changing conditional statements that are time-dependent. Miners have the ability to adjust timestamps slightly, which can prove to be dangerous if block timestamps are used incorrectly in smart contracts.

The Attack

Consider the Roulette contract which is just a normal lottery game. The *block.timestamp* and its alias *now* can be manipulated by miners if they have some incentive to do so.

```
1  contract Roulette {
2      uint public pastBlockTime;
3
4      constructor() public payable {}
5
6      function () public payable {
7          require(msg.value == 10 ether);
8          require(now != pastBlockTime);
9          pastBlockTime = now;
10         if(now % 15 == 0) {
11             msg.sender.transfer(this.balance);
12         }
13     }
14 }
```

Listing 4.12 Roulette contract

One transaction per block can bet 10 ether for a chance to win the balance of the contract. The assumption here is that the last two digits of *block.timestamp* are uniformly distributed. If that were the case, there would be a 1/15 chance of winning this lottery.

However, miners can adjust the timestamp should they need to. In this particular case, if enough ether pools in the contract, a miner who solves a block is incentivized to choose a timestamp such that *block.* or *now modulo 15 is 0.* In doing, so they may win the ether locked in this contract along with the block reward. As there is only one person allowed to bet per block, this is also vulnerable to front-running attacks.

In practice, block timestamps are monotonically increasing and so miners cannot choose arbitrary block timestamps. They are also limited to setting block times not too far in the future, as these blocks will likely be rejected by the network.

Preventative Technique

Block timestamps should not be used for entropy or generating random numbers. They should not be the deciding factor for winning a game or changing an important state.

Time sensitive logic is sometimes required. For unlocking contracts (time-locking), completing an ICO after a few weeks, or enforcing expiry dates. It is sometimes recommended to use block.number and an average block time to estimate times; with a 10-second block time, 1 week equates to approximately, 60480 blocks. Thus, specifying a block number at which to change a contract state can be more secure, as miners are unable easily to manipulate the block number. The BAT ICO contract employed this strategy.

This can be unnecessary if contracts are not particularly concerned with miner manipulations of the block timestamp, but it is something to be aware of when developing contracts.

Chapter 5

Ethereum Smart Contract Best Practices

Like most complex Blockchain programs, Ethereum is new and extremely experimental. Therefore, we should expect constant changes in the security landscape, as new bugs and security risks are discovered, and new best practices are developed.

Smart contract programming requires a different engineering mindset than we may be used to. The cost of failure can be high, and change can be difficult, making it in some ways more similar to hardware programming or financial services programming than web or mobile development. It is therefore not enough to defend against known vulnerabilities. Instead, we will need to learn a new philosophy of development.

5.1 Secure Development Recommendations

5.1.1 External Calls

Use caution when making external calls

Calls to untrusted contracts can introduce several unexpected risks or errors. External calls may execute malicious code in that contract or any other contract that it depends upon. As such, every external call should be treated as a potential security risk. When it is not possible, or undesirable to remove external calls, use the recommendations in the rest of this section to minimize the danger.

Mark untrusted contracts

When interacting with external contracts, a good practice is to name all the variables, methods, and contract interfaces in a way that makes it clear that interacting with them is potentially unsafe. This applies to your own functions that call external contracts.

The code of Listing 6.6 demonstrates a bad practice of marking.

```
1 Bank.withdraw(100);
2
3 function makeWithdrawal(uint amount) {
4     Bank.withdraw(amount);
5 }
```

Listing 5.1 Bad marking practice

Instead, the code of Listing 5.2 demonstrates a good practice of marking.

```
1 UntrustedBank.withdraw(100); //untrusted external call
2 TrustedBank.withdraw(100); //external but trusted bank contract
3
4 function makeUntrustedWithdrawal(uint amount) {
5     UntrustedBank.withdraw(amount);
6 }
```

Listing 5.2 Good marking practice

Avoid state changes after external calls

Whether *raw calls* or *contract calls* are used, we have to assume that malicious code might be executed. Even if an external contract is not malicious, malicious code can be executed by any contracts it calls.

One particular danger is malicious code may hijack the control flow, leading to vulnerabilities due to reentrancy.

If you are making a call to an untrusted external contract, avoid state changes after the call. This pattern is also sometimes known as the checks-effects-interactions pattern.

Do not use *transfer* or *send* functions

The *transfer* and *send* functions forward exactly 2300 gas to the recipient. The goal of this hard-coded gas stipend was to prevent reentrancy vulnerabilities, but this only makes sense under the assumption that gas costs are constant.

Recently EIP 1884 [21] was included in the Istanbul hard fork. One of the changes included in EIP 1884 is an increase to the gas cost of the SLOAD operation, causing a contract's fallback function to cost more than 2300 gas.

It is recommended to stop using *transfer* and *send* and instead use *call*.

In Figure 5.3 is presenting an example of using the *transfer* function. However, the code in Figure 5.4 is more secure, because of the replacement of the *transfer* function.

```
1 contract Vulnerable {
2     function withdraw(uint256 amount) external {
3         msg.sender.transfer(amount); //Forwards 2300 gas, which may
              not be enough if the recipient is a contract and gas costs
               change
4     }
5 }
```

Listing 5.3 Using the *transfer* function

```
1 contract Fixed {
2     function withdraw(uint256 amount) external {
3         //forwards all available gas
4         (bool success, ) = msg.sender.call.value(amount)("");
5         require(success, "Transfer failed.");
6     }
7 }
```

Listing 5.4 Replacing the *transfer* function

Note that *call* function does nothing to mitigate reentrancy attacks, so other precautions must be taken. To prevent reentrancy attacks, it is recommended that you use the checks-effects-interactions pattern.

Handle errors in external calls

Solidity offers low-level call methods that work on raw addresses such as *address.call*, *address.callcode*, *address.delegatecal*, and *address.send*. These low-level methods never throw an exception, but will return false if the call encounters an exception. On the other hand, contract calls will automatically propagate a throw.

If you choose to use the low-level call methods, make sure to handle the possibility that the call will fail, by checking the return value.

Figure 5.5 is presenting a piece of code which does not handle errors.

```
1 someAddress.send(55);
2 someAddress.call.value(55)(""); //it will forward all remaining gas
    and doesn't check for result
3
4 someAddress.call.value(100)(bytes4(sha3("deposit()"))); //if deposit
    throws an exception, the raw call() will only return false and
    transaction will not be reverted
```

Listing 5.5 Code with no-error handler

On the other hand, Figure 5.6 contains an error handler.

```
1 (bool success, ) = someAddress.call.value(55)("");
2 if(!success) {
3     // handle failure code
4 }
5
6 ExternalContract(someAddress).deposit.value(100)();
```

Listing 5.6 Code with error handler

Favor pull over push for external calls

External calls can fail accidentally or deliberately. To minimize the damage caused by such failures, it is often better to isolate each external call into its own transaction that can be initiated by the recipient of the call. This is especially relevant for payments, where it is better to let users withdraw funds rather than push funds to them automatically. Avoid combining multiple ether transfers in a single transaction.

Below it is shown a multiple ether transfer contract and also a non-multiple ether transfer contract.

```
1  contract auction {
2      address highestBidder;
3      uint highestBid;
4
5      function bid() payable {
6          require(msg.value >= highestBid);
7
8          if (highestBidder != address(0)) {
9              (bool success, ) = highestBidder.call.value(highestBid)("
                   ");
10             require(success);
11             //if this call consistently fails, no one else can bid
12         }
13
14        highestBidder = msg.sender;
15        highestBid = msg.value;
16     }
17 }
```

Listing 5.7 Multiple ether transfer contract

```
1  contract auction {
2      address highestBidder;
3      uint highestBid;
4      mapping(address => uint) refunds;
5
6      function bid() payable external {
7          require(msg.value >= highestBid);
8
9          if (highestBidder != address(0)) {
10             //record the refund that this user can claim
11             refunds[highestBidder] += highestBid;
12         }
13
```

```
14          highestBidder = msg.sender;
15          highestBid = msg.value;
16      }
17
18      function withdrawRefund() external {
19          uint refund = refunds[msg.sender];
20          refunds[msg.sender] = 0;
21          (bool success, ) = msg.sender.call.value(refund)("");
22          require(success);
23      }
24  }
```

Listing 5.8 Non multiple ether transfer contract

Do not delegatecall to untrusted code

The *delegatecall* function is used to call functions from other contracts as if they belong to the caller contract. Thus, the caller may change the state of the calling address. This may be insecure.

An example below shows how using delegatecall can lead to the destruction of the contract and loss of its balance.

```
1   contract Destructor {
2       function doWork() external {
3           selfdestruct(0);
4       }
5   }
6
7   contract Worker {
8       function doWork(address _internalWorker) public {
9           //unsafe
10          _internalWorker.delegatecall(bytes4(keccak256("doWork()")));
11      }
12  }
```

Listing 5.9 Unsafe usage of delegatecall

If *Worker.doWork* is called with the address of the deployed Destructor contract as an argument, the Worker contract will self-destruct. Delegate execution only to trusted contracts, and never to a user supplied address.

5.1.2 Fallback Function

Keep fallback functions simple

Fallback function is called when a contract is sent a message with no arguments, and only has access to 2300 gas when called from a *send* or *transfer* function. If you wish to be able to receive ether from a *send* or *transfer* function, the most you can do in a *fallback* function is to log an event. Use a proper function if a computation of more gas is required.

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

Listing 5.10 Bad practice for using fallback function

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

function() payable {
    require(msg.data.length == 0);
    emit LogDepositReceived(msg.sender);
}
```

Listing 5.11 Good practice for using fallback function

Check data length in fallback functions

Since the *fallback* function is not only called for plain ether transfers (without data) but also when no other function matches, you should check that the data is empty if the fallback function is intended to be used only for the purpose of logging received ether. Otherwise, callers will not notice if your contract is used incorrectly and functions that do not exist are called.

```
function() payable {
    emit LogDepositReceived(msg.sender);
}
```

Listing 5.12 Fallback function without data length check

```
function() payable {
    require(msg.data.length == 0);
    emit LogDepositReceived(msg.sender);
}
```

Listing 5.13 Fallback function with data length check

5.1.3 Timestamp Dependence

There are three main considerations when using a timestamp to execute a critical function in a contract, especially when actions involve fund transfer.

Timestamp Manipulation

Be aware that the timestamp of the block can be manipulated by a miner. Consider the following contract:

```
uint256 constant private salt =  block.timestamp;

function random(uint Max) constant private returns (uint256 result){
    //get the best seed for randomness
    uint256 x = salt * 100/Max;
    uint256 y = salt * block.number/(salt % 5) ;
    uint256 seed = block.number/3 + (salt % 300) + Last_Payout + y;
    uint256 h = uint256(block.blockhash(seed));

    //random number between 1 and max
    return uint256((h / x)) % max + 1;
}
```

Listing 5.14 Random generated timestamp

When the contract uses the timestamp to seed a random number, the miner can actually post a timestamp within 15 seconds of the block being validated, effectively allowing the miner to precompute an option more favorable to their chances in the lottery. Timestamps are not random and should not be used in that context.

The 15-second Rule

Ethereum's Yellow Paper [24] does not specify a constraint on how many blocks can drift in time, but it does specify that each timestamp should be bigger than the timestamp of its parent. Popular Ethereum protocol implementations Geth [5] and Parity [6] both reject blocks with timestamp more than 15 seconds in the future. Therefore, a good rule of thumb in evaluating timestamp usage is that if the scale of your time-dependent event can vary by 15 seconds and maintain integrity, it is safe to use a *block.timestamp*.

Avoid using *block.number* as a timestamp

It is possible to estimate a time delta using the block.number property and average block time, however this is not future proof as block times may change (such as fork reorganizations and the difficulty bomb). In a sale spanning days, the 15-second rule allows one to achieve a more reliable estimate of time.

5.2 Software Engineering Techniques

There are many attacks against smart contracts, so trying to protect them becomes more difficult. Since the cost of failure on a Blockchain can be very high, we must also adapt the way we write software, to account for that risk.

A popular approach is to prepare for failure. It is impossible to know in advance whether our code is secure. However, we can architect our contracts in a way that allows them to fail gracefully, and with minimal damage. This section presents a variety of techniques that will help us prepare for failure.

However, there's always a risk when we add a new component to our system. A badly designed fail-safe could itself become a vulnerability as can the interaction between a number of well-designed fail-safes. We have to be thoughtful about each technique you use in our contracts, and consider carefully how they work together to create a robust system.

5.2.1 Upgrading Broken Contracts

Code will need to be changed if errors are discovered or if improvements need to be made. It is no good to discover a bug, but have no way to deal with it.

There are two basic approaches that are most commonly used. The simpler of the two is to have a registry contract that holds the address of the latest version of the contract. A more seamless approach for contract users is to have a contract that forwards calls and data onto the latest version of the contract.

Whatever the technique, it is important to have modularization and good separation between components, so that code changes do not break functionality, orphan data, or require substantial costs to port. In particular, it is usually beneficial to separate complex logic from data storage, so that you do not have to recreate all the data in order to change the functionality.

It is also critical to have a secure way for parties to decide to upgrade the code. Depending on contract, code changes may need to be approved by a single trusted party, a group of members, or a vote of the full set of stakeholders. If this process can take some time, you will want to consider if there are other ways to react more quickly in case of an attack, such as an emergency stop or circuit-breaker.

Regardless of your approach, it is important to have some way to upgrade contracts, or they will become unusable when the inevitable bugs are discovered in them.

Use a registry contract to store latest version of a contract

In the following example, the calls are not forwarded, so users should fetch the current address each time before interacting with it.

```
1  contract SomeRegister {
2      address backendContract;
3      address[] previousBackends;
4      address owner;
5
6      constructor() {
7          owner = msg.sender;
8      }
9
10     modifier onlyOwner() {
11         require(msg.sender == owner)
12         _;
13     }
```

```
14
15      function changeBackend(address newBackend) public
16      onlyOwner()
17      returns (bool)
18      {
19          if(newBackend != address(0) && newBackend != backendContract)
                {
20              previousBackends.push(backendContract);
21              backendContract = newBackend;
22              return true;
23          }
24          return false;
25      }
26  }
```

Listing 5.15 SomeRegister Contract

There are two main disadvantages to this approach:

- Users must always look up the current address, and anyone who fails to do so risks using an old version of the contract.
- You will need to think carefully about how to deal with the contract data when you replace the contract.

Use a *delegatecall* to forward data and calls

This approach relies on using the fallback function (in Relay contract) to forward the calls to a target contract (LogicContract) using delegatecall. The delegatecall is a special function in Solidity that executes the logic of the called address (LogicContract) in the context of the calling contract (Relay), so "storage, current address and balance still refer to the calling contract, only the code is taken from the called address".

```
1  contract Relay {
2      address public currentVersion;
3      address public owner;
4
5      modifier onlyOwner() {
6          require(msg.sender == owner);
7          _;
8      }
9
```

```
10      constructor(address initAddr) {
11          require(initAddr != address(0));
12          currentVersion = initAddr;
13          owner = msg.sender; //this owner may be another contract with
                multisig, not a single contract owner
14      }
15
16      function changeContract(address newVersion) public
17      onlyOwner()
18      {
19          require(newVersion != address(0));
20          currentVersion = newVersion;
21      }
22
23      fallback() external payable {
24          (bool success, ) = address(currentVersion).delegatecall(msg.
               data);
25          require(success);
26      }
27  }
```

Listing 5.16 Relay Contract

```
1   contract LogicContract {
2       address public currentVersion;
3       address public owner;
4       uint public counter;
5
6       function incrementCounter() {
7           counter++;
8       }
9   }
```

Listing 5.17 LogicContract Contract

This simple version of the pattern cannot return values from LogicContract's functions, only forward them, which limits its applicability. More complex implementations attempt to solve this with in-line assembly code and a registry of return sizes. They are commonly referred to as Proxy Patterns, but are also known as Router, Dispatcher and Relay. Each implementation variant introduces a different set of complexity, risks and limitations.

You must be extremely careful with how you store data with this method. If your new contract has a different storage layout than the first, your data may end up corrupted. When using more complex implementations of delegatecall, you should carefully consider and understand the following:

- How the EVM handles the layout of state variables in storage, including packing multiple variables into a single storage slot if possible.
- How and why the order of inheritance impacts the storage layout.
- Why the called contract must have the same storage layout of the calling contract, and only append new variables to the storage.
- Why a new version of the called contract must have the same storage layout as the previous version, and only append new variables to the storage.
- How a contract's constructor can affect upgradeability.
- How the ABI specifies function selectors and how function-name collision can be used to exploit a contract that uses delegatecall.
- How delegatecall to a non-existent contract will return true even if the called contract does not exist.
- The importance of immutability to achieve truslessness.

5.2.2 Circuit Breakers (Pause contract functionality)

Circuit breakers stop execution if certain conditions are met, and can be useful when new errors are discovered. For example, most actions may be suspended in a contract if a bug is discovered, and the only action now active is a withdrawal. You can either give certain trusted parties the ability to trigger the circuit breaker or else have programmatic rules that automatically trigger the certain breaker when certain conditions are met. An example is shown below.

```
1  bool private stopped = false;
2  address private owner;
3
4  modifier isAdmin() {
5      require(msg.sender == owner);
6      _;
7  }
8
9  function toggleContractActive() isAdmin public {
10     //you can add an additional modifier that restricts stopping a
           contract to be based on another action, such as a vote of
           users
11     stopped = !stopped;
```

```
12 }
13
14 modifier stopInEmergency { if (!stopped) _; }
15 modifier onlyInEmergency { if (stopped) _; }
16
17 function deposit() stopInEmergency public {
18     //some code
19 }
20
21 function withdraw() onlyInEmergency public {
22     //some code
23 }
```

Listing 5.18 Pause contract functionality

5.2.3 Speed Bumps (Delay contract actions)

Speed bumps slow down actions, so that if malicious actions occur, there is time to recover. For example, The DAO [19] required 27 days between a successful request to split the DAO and the ability to do so. This ensured the funds were kept within the contract, increasing the likelihood of recovery. In the case of the DAO, there was no effective action that could be taken during the time given by the speed bump, but in combination with our other techniques, they can be quite effective.

```
1  struct RequestedWithdrawal {
2      uint amount;
3      uint time;
4  }
5
6  mapping (address => uint) private balances;
7  mapping (address => RequestedWithdrawal) private requestedWithdrawals
       ;
8  uint constant withdrawalWaitPeriod = 28 days; //4 weeks
9
10 function requestWithdrawal() public {
11     if (balances[msg.sender] > 0) {
12         uint amountToWithdraw = balances[msg.sender];
13         balances[msg.sender] = 0; //withdraw everything
14         //the deposit function prevents new deposits when withdrawals
               are in progress
15
16         requestedWithdrawals[msg.sender] = RequestedWithdrawal({
```

```
            amount: amountToWithdraw,
            time: now
        });
    }
}

function withdraw() public {
    if(requestedWithdrawals[msg.sender].amount > 0 && now >
       requestedWithdrawals[msg.sender].time + withdrawalWaitPeriod)
       {
        uint amountToWithdraw = requestedWithdrawals[msg.sender].
           amount;
        requestedWithdrawals[msg.sender].amount = 0;

        require(msg.sender.send(amountToWithdraw));
    }
}
```

Listing 5.19 Delay contract actions

5.2.4 Rate Limiting

Rate limiting halts or requires approval for substantial changes. For example, a depositor may only be allowed to withdraw a certain amount or percentage of total deposits over a certain time period (e.g., max 100 ether over 1 day) additional withdrawals in that time period may fail or require some sort of special approval. Or the rate limit could be at the contract level, with only a certain amount of tokens issued by the contract over a time period.

```
uint internal period; //how many blocks before limit resets
uint internal limit; //max ether to withdraw per period
uint internal currentPeriodEnd; //block which the current period ends
    at
uint internal currentPeriodAmount; //amount already withdrawn this
   period

constructor(uint _period, uint _limit) public {
    period = _period;
    limit = _limit;

    currentPeriodEnd = block.number + period;
}

```

```
13 function withdraw(uint amount) public {
14     //update period before proceeding
15     updatePeriod();
16
17     //prevent overflow
18     uint totalAmount = currentPeriodAmount + amount;
19     require(totalAmount >= currentPeriodAmount, 'overflow');
20
21     //disallow withdraws that exceed current rate limit
22     require(currentPeriodAmount + amount < limit, 'exceeds period
           limit');
23     currentPeriodAmount += amount;
24     msg.sender.transfer(amount);
25 }
26
27 function updatePeriod() internal {
28     if(currentPeriodEnd < block.number) {
29         currentPeriodEnd = block.number + period;
30         currentPeriodAmount = 0;
31     }
32 }
```

Listing 5.20 Rate Limiting

5.2.5 Contract Rollout

Contracts should have a substantial and prolonged testing period, before substantial money is put at risk. At minimum, you should:

- Have a full test suite with 100
- Deploy on your own testnet.
- Deploy on the public testnet with substantial testing and bug bounties.
- Exhaustive testing should allow various players to interact with the contract at volume.
- Deploy on the mainnet in beta, with limits to the amount at risk.

During testing, you can force an automatic deprecation by preventing any actions, after a certain time period. For example, an alpha contract may work for several weeks and then automatically shut down all actions, except for the final withdrawal.

```
modifier isActive() {
    require(block.number <= SOME_BLOCK_NUMBER);
    _;
}

function deposit() public isActive {
    //some code
}

function withdraw() public {
    //some code
}
```

Listing 5.21 Rollout

In the early stages, you can restrict the amount of Ether for any user reducing the risk.

Chapter 6

Real World Attack Examples

6.1 The DAO

The DAO (Decentralized Autonomous Organization) attack was one of the major hacks that occurred in the early development of Ethereum. At the time, the contract held over 150 million dollars. Reentrancy played a major role in the attack, which ultimately led to the hard fork that created ETC (Ethereum Classic).

6.2 PoWHC and Batch Transfer Overflow (CVE-2018–10299)

PoWHC (Proof of Weak Hands Coin) [7], originally devised as a joke of sorts, was a Ponzi scheme [8] written by an internet collective. Unfortunately it seems that the author of the contract had not seen over/underflows before, and consequently 866 ether were liberated from its contract. Eric Banisadr gives a good overview of how the underflow occurred (which is not too dissimilar to the Ethernaut challenge described earlier) in his blog post on the event.

Another example comes from the implementation of a $batchTransfer$ function into a group of ERC20 token contracts. The implementation contained an overflow vulnerability.

6.3 Parity Multisig Wallet (Second Hack))

The Second Parity Multisig Wallet hack is an example of how well-written library code can be exploited if run outside its intended context. There are a number of good explanations of this hack. The WalletLibrary and WalletEvents contract are shown below:

```
contract WalletLibrary is WalletEvents {

    //throw unless the contract is not yet initialized.
    modifier only_uninitialized { if (m_numOwners > 0) throw; _; }

    //constructor
    function initWallet(address[] _owners, uint _required, uint
        _daylimit)
         only_uninitialized {
         initDaylimit(_daylimit);
         initMultiowned(_owners, _required);
    }

    //kills the contract sending everything to `_to`.
    function kill(address _to) onlymanyowners(sha3(msg.data))
        external {
         suicide(_to);
    }
}
```

Listing 6.1 WalletLibrary Contract

And below is the Wallet contract:

```
contract Wallet is WalletEvents {

    //gets called when no other function matches
    function() payable {
        //just being sent some cash?
        if (msg.value > 0)
            Deposit(msg.sender, msg.value);
        else if (msg.data.length > 0)
            _walletLibrary.delegatecall(msg.data);
        }

    //fields
    address constant _walletLibrary =
        0xcafecafecafecafecafecafecafecafecafecafe;
 }
```

Listing 6.2 Wallet Contract

Notice that the Wallet contract essentially passes all calls to the WalletLibrary contract via a delegatecall. The constant *_walletLibrary* address in this code snippet acts as a placeholder for the actually deployed WalletLibrary contract. The intended operation of these contracts was to have a simple low-cost deployable Wallet contract whose codebase

and main functionality were in the WalletLibrary contract. Unfortunately, the WalletLibrary contract is itself a contract and maintains its own state.

It is possible to send calls to the WalletLibrary contract itself. Specifically, this contract could be initialized and become owned. In fact, a user did this, calling the initWallet function on the WalletLibrary contract and becoming an owner of the library contract. The same user subsequently called the *kill* function. Because the user was an owner of the library contract, the modifier passed and the library contract self-destructed. As all Wallet contracts in existence refer to this library contract and contain no method to change this reference, all of their functionality, including the ability to withdraw ether, was lost along with the WalletLibrary contract. As a result, all ether in all Parity multisig wallets of this type instantly became lost or permanently unrecoverable.

6.4 Parity Multisig Wallet (First Hack)

In the first Parity multisig hack, about 31 million dollars worth of Ether was stolen, mostly from three wallets. A good recap of exactly how this was done is given by Haseeb Qureshi. Essentially, the multisig wallet is constructed from a base Wallet contract, which calls a library contract containing the core functionality. The WalletLibrary contract contains the code to initialize the wallet, as can be seen from the following snippet:

```
1  contract WalletLibrary is WalletEvents {
2
3      //constructor is given number of sigs required to do protected
4      //"onlymanyowners" transactions as well as the selection of
           addresses capable of confirming them
5      function initMultiowned(address[] _owners, uint _required) {
6          m_numOwners = _owners.length + 1;
7          m_owners[1] = uint(msg.sender);
8          m_ownerIndex[uint(msg.sender)] = 1;
9          for (uint i = 0; i < _owners.length; ++i) {
10             m_owners[2 + i] = uint(_owners[i]);
11             m_ownerIndex[uint(_owners[i])] = 2 + i;
12         }
13         m_required = _required;
14     }
15
16     // constructor - just pass on the owner array to multiowned and
17     // the limit to daylimit
```

```
18      function initWallet(address[] _owners, uint _required, uint
            _daylimit) {
19          initDaylimit(_daylimit);
20          initMultiowned(_owners, _required);
21      }
22  }
```

Listing 6.3 WalletLibrary Contract

Note that neither of the functions specifies their visibility, so both default to public. The *initWallet* function is called in the wallet's constructor, and sets the owners for the multisig wallet as can be seen in the *initMultiowned* function. Because these functions were accidentally left public, an attacker was able to call these functions on deployed contracts, resetting the ownership to the attacker's address. Being the owner, the attacker then drained the wallets of all their ether.

6.5 PRNG Contracts

In February 2018 Arseny Reutov blogged about his analysis of 3,649 live smart contracts that were using some sort of pseudorandom number generator (PRNG) [4]. He found 43 contracts that could be exploited.

6.6 Reentrancy Honey Pot

A number of recent honey pots have been released on the mainnet. These contracts try to outsmart Ethereum hackers who try to exploit the contracts, but who in turn end up losing ether to the contract they expect to exploit. One example employs this attack by replacing an expected contract with a malicious one in the constructor.

```
1   contract Private_Bank {
2       mapping (address => uint) public balances;
3       uint public MinDeposit = 1 ether;
4       Log TransferLog;
5
6       function Private_Bank(address _log) {
7           TransferLog = Log(_log);
8       }
9
10      function Deposit() public payable {
11          if(msg.value >= MinDeposit) {
```

```
12              balances[msg.sender]+=msg.value;
13              TransferLog.AddMessage(msg.sender,msg.value,"Deposit");
14          }
15      }
16
17      function CashOut(uint _am) {
18          if(_am<=balances[msg.sender]) {
19              if(msg.sender.call.value(_am)()) {
20                  balances[msg.sender]-=_am;
21                  TransferLog.AddMessage(msg.sender,_am,"CashOut");
22              }
23          }
24      }
25
26      function() public payable{}
27  }
28
29  contract Log {
30      struct Message {
31          address Sender;
32          string Data;
33          uint Val;
34          uint Time;
35      }
36
37      Message[] public History;
38      Message LastMsg;
39
40      function AddMessage(address _adr,uint _val,string _data) public
                {
41          LastMsg.Sender = _adr;
42          LastMsg.Time = now;
43          LastMsg.Val = _val;
44          LastMsg.Data = _data;
45          History.push(LastMsg);
46      }
47  }
```

Listing 6.4 Private_Bank and Log Contract

This post by one reddit user explains how they lost 1 ether to this contract by trying to exploit the reentrancy bug they expected to be present in the contract.

6.7 Etherpot and King of the Ether

Etherpot was a lottery smart contract. The downfall of this contract was primarily due to incorrect use of block hashes (only the last 256 block hashes are usable). However, this contract also suffered from an unchecked call value. Consider the following function:

```
function cash(uint roundIndex, uint subpotIndex){
    var subpotsCount = getSubpotsCount(roundIndex);

    if(subpotIndex>=subpotsCount)
        return;

    var decisionBlockNumber = getDecisionBlockNumber(roundIndex,
       subpotIndex);

    if(decisionBlockNumber>block.number)
        return;

    if(rounds[roundIndex].isCashed[subpotIndex])
        return;

    //subpots can only be cashed once. This is to prevent double
       payouts

    var winner = calculateWinner(roundIndex,subpotIndex);
    var subpot = getSubpot(roundIndex);

    winner.send(subpot);

    rounds[roundIndex].isCashed[subpotIndex] = true;
    //mark the round as cashed
}
```

Listing 6.5 WalletLibrary Contract

6.8 ERC20 and Bancor

The ERC20 [18] standard is quite well-known for building tokens on Ethereum. This standard has a potential front-running vulnerability that comes about due to the approval function. Mikhail Vladimirov and Dmitry Khovratovich have written a good explanation of this vulnerability (and ways to mitigate the attack). The standard specifies the approval function as:

```
1 function approve(address _spender, uint256 _value) returns (bool
      success) {}
```

Listing 6.6 Approve function

This function allows a user to permit other users to transfer tokens on their behalf. The front-running vulnerability occurs in the scenario where a user Alice approves her friend Bob to spend 100 tokens. Alice later decides that she wants to revoke Bob's approval to spend, say, 100 tokens, so she creates a transaction that sets Bob's allocation to 50 tokens. Bob, who has been carefully watching the chain, sees this transaction and builds a transaction of his own spending the 100 tokens. He puts a higher gasPrice on his transaction than Alice's, so gets his transaction prioritized over hers. Some implementations of approve would allow Bob to transfer his 100 tokens and then, when Alice's transaction is committed, reset Bob's approval to 50 tokens, in effect giving Bob access to 150 tokens.

Another prominent real-world example is Bancor. Ivan Bogatyy and his team documented a profitable attack on the initial Bancor implementation. His blog post and DevCon3 talk discuss in detail how this was done. Essentially, prices of tokens are determined based on transaction value; users can watch the transaction pool for Bancor transactions and front-run them to profit from the price differences. This attack has been addressed by the Bancor team.

6.9 GovernMental

GovernMental was an old Ponzi scheme that accumulated quite a large amount of ether (1,100 ether, at one point). Unfortunately, it was susceptible to the DoS vulnerabilities mentioned in this section. A Reddit post by etherik describes how the contract required the deletion of a large mapping in order to withdraw the ether. The deletion of this mapping had a gas cost that exceeded the block gas limit at the time, and thus it was not possible to withdraw the 1,100 ether. The contract address is:

`0xF45717552f12Ef7cb65e95476F217Ea008167Ae3`

and you can see from transaction:

`0a1b0793e8ec4fc105257e8128f0506b`

that the 1,100 ether were finally obtained with a transaction that used 2.5M gas (when the block gas limit had risen enough to allow such a transaction).

GovernMental, the old Ponzi scheme mentioned above, was also vulnerable to a timestamp-based attack. The contract paid out to the player who was the last player to join (for at least one minute) in a round. Thus, a miner who was a player could adjust the timestamp (to a future time, to make it look like a minute had elapsed) to make it appear that they were the last player to join for over a minute (even though this was not true in reality).

6.10 Rubixi

Rubixi [17] was another pyramid scheme that exhibited this kind of vulnerability. It was originally called DynamicPyramid, but the contract name was changed before deployment to Rubixi. The constructor's name wasn't changed, allowing any user to become the creator. Some interesting discussion related to this bug can be found on Bitcointalk. Ultimately, it allowed users to fight for creator status to claim the fees from the pyramid scheme.

6.11 OpenAddressLottery and CryptoRoulette Honey Pots

A honey pot named *OpenAddressLottery* [1] was deployed that used this uninitialized storage variable quirk to collect ether from some would-be hackers. The contract is rather involved, so we will leave the analysis to the Reddit thread where the attack is quite clearly explained. Another honey pot, *CryptoRoulette* [20], also utilized this trick to try to collect some ether.

6.12 Ethstick

The *Ethstick* contract does not use extended precision; however, it deals with wei. So, this contract will have issues of rounding, but only at the wei level of precision. It has some more serious flaws, but these relate back to the difficulty in getting entropy on the blockchain.

Chapter 7

Challenges and Opportunities

7.1 Security Design

With the application of smart contracts in different industries, more security patterns and security anti-patterns should be extracted from the emerging security vulnerabilities.

Security Modeling is a promising research direction. There is still considerable room for this direction. In FSMs-based approaches [2], the number of states and transitions grows exponentially with the number of contracts, which makes this modeling approach inappropriate for complex business logic. Moreover, a logic-based approach is still in the proof-of-concept stage even if this approach has distinct advantages in negotiation, notarizing, and enforcement of a contract. The algorithm for a logic approach is not efficient in blockchain scenarios. Moreover, how to effectively extract FSMs and logic rules from legal agreements to mitigate vulnerabilities is worth exploring. How to make the logic and procedural approaches compatible is also challenging.

With the popularity of blockchain, more domain-specific modeling approaches and logic languages with precise semantics should be proposed to avoid vulnerabilities. These models and languages should be easy to convert into code or real legal contracts adopted by the court directly.

7.2 Security Implementation

A promising research direction in security implementation of smart contracts is the integration of more security libraries, such as OpenZeppelin, into the contract development environment in the form of security plugins. These libraries will provide security enhancement for the development of smart contracts.

Another promising research direction in security implementation of smart contracts is the development of a formal high-level domain-specific language with explicit semantics and security enhancements. With this language, developers could write contract templates, from which legal agreements and executable smart contract code could be automatically extracted, thereby avoiding error-prone manual coding and eliminating major security issues. Moreover, to keep templates simple and semantics precise, developers could also integrate logic rules in a logic-based approach into template technology.

7.3 Testing Before Deployment

Rigorous formal verification is an effective approach for detecting security vulnerabilities. For Ethereum, rigorous formal verification formalizes the EVM interpreter or smart contract code using a general-purpose intermediate-level language, such as Lem. This language serves as an intermediate between high-level languages and executable bytecode. That is, contract code in high-level languages will first be compiled into contract code in intermediate-level languages instead of EVM bytecode. Because these intermediate-level languages are well integrated with the program prover, it is amenable to the program verification. A promising research direction is to design new intermediate-level languages specifically for smart contracts, instead of these general-purpose intermediate-level languages, to facilitate the formal verification of smart contracts. Scilla [12] is such an attempt. With the popularity of the Ethereum platform, more and more such intermediate-level languages that are easy to understand but that have strong expressive power and precise operational semantics will appear. Different from Ethereum, Fabric [13] uses a general-purpose programming language to write smart contracts. Accordingly, the formalization of smart contracts for Fabric may be more complicated than that for Ethereum. Fortunately, some provers associated with general programming languages are available, and how to adapt these provers to blockchain scenarios is also worth exploring.

In addition to rigorous formal verification, code analysis tools are also exploited to find security vulnerabilities in smart contracts. However, these tools are still in the infancy stage. Their performance is not ideal. Many of them do not cover the full range of security

vulnerabilities outlined in this paper. Moreover, there is still considerable room to improve their accuracy, false-positive rate, and false-negative rate in the process of vulnerability detection. For this aspect, deep learning may be useful. Moreover, because new security vulnerabilities continue to emerge, an excellent code analysis tool should be extensible. The tool should be able to identify new security vulnerabilities by defining new security properties. However, security properties are now often specified ad hoc and are mostly verified manually. A method for the unified and flexible specification of security properties to facilitate automatic verification is urgently needed.

In brief, because smart contracts are hard to patch once they are deployed on the blockchain, testing before deployment is significant. A convenient test platform for smart contracts, especially new test techniques such as mocking objects capable of effectively emulating the blockchain, are also worth exploring further. Additionally, the construction of standard test datasets for a specific blockchain platform to mitigate the difficulty of testing smart contracts is also a promising research direction.

7.4 Monitoring and Analysis

Real-time monitoring and analysis are very significant for blockchain security. Blockchain platforms, such as Ethereum and Fabric, are expected to provide an internal framework for efficient monitoring of the execution of smart contracts in the future. Based on such a framework, organizations can expand the scope of security monitoring according to their business.

As more and more transactions of different businesses are dealt with on the blockchain, a variety of attacks and scams will emerge due to the enormous value transfer associated with these transactions, which provides a promising research direction for data mining on the blockchain. For instance, we may discover more new insights regarding anomalous behavior in smart contracts by in-depth graph analysis.

7.5 Other Directions

If the security of smart contracts is considered at the system level we will spend less on security in the contract development lifecycle. However, consideration of security at the system level is a more general problem and thus is also a significant challenge.

Corresponding to traditional software, some researchers advocate for a discipline of blockchain software engineering to address the issues posed by smart contract programming. We also hold this viewpoint. With the progress of blockchain security, blockchain engineers and researchers may propose more new best practices and security tools concerning blockchain software engineering.

Chapter 8

Conclusion

As examined in this thesis, the environment in which smart contracts are executed is decentralized and hard to patch for bugs. Consequently, smart contracts have many security issues in terms of security vulnerabilities and anomalous activities. Our focus on smart contract security from the perspective of the software lifecycle enabled us to reveal the practical and theoretical aspects of smart contract security to a greater degree than any previous study. We achieved this thesis outcome as follows.

First, we illustrated the key features of blockchains including:

- P2P Network, the main network architecture of blockchains
- Consensus Rules, which are the most important processes in order to achieve consensus between unreliable parties
- Chain of Blocks (Blockchain). A linked structure where all transactions are kept
- Consensus Algorithm, specific consensus rules for each blockchain

We also presented Ethereum Blockchain components by explaining:

- Transactions, the main assets in the Ethereum network
- Ethereum Virtual Machine (EVM), a world computer which compiles and executes code for Ethreum Smart Contracts
- Clients, the participants of the Ethereum network

Then we introduced the concept of Smart Contract and revealed the most common security vulnerabilities of them in Ethereum network. After that, we demystified the puzzles of security solutions for smart contracts in terms of numerous security themes spanning four contract development phases:

- Secure Development Recommendations
- Software Engineering Techniques

Finally, we summarized the challenges and opportunities of security research for smart contracts as a software engineering problem, and suggested several research directions to improve smart contract security. I hope that this thesis builds on the current research outcomes of smart contract security and signifies a milestone in information security in the age of blockchain. I expect that many more research efforts will follow by expanding on the works classified in this thesis and by applying the techniques outlined here to various business contexts.

References

[1] Openaddresslottery. Feb 10, 2018. URL https://etherscan.io/address/0x741f1923974464efd0aa70e77800ba5d9ed18902#code.

[2] Aron Laszka Anastasia Mavridou. Designing secure ethereum smart contracts: A finite state machine based approach. Dec, 2018. URL https://www.researchgate.net/publication/335479007_Designing_Secure_Ethereum_Smart_Contracts_A_Finite_State_Machine_Based_Approach.

[3] Dave Appleton. Ethereum multi-signature wallets. Jun 26, 2017. URL https://medium.com/hellogold/ethereum-multi-signature-wallets-77ab926ab63b.

[4] David Bertoldi. Building a pseudorandom number generator. Nov 11, 2019. URL https://towardsdatascience.com/building-a-pseudorandom-number-generator-9bc37d3a87d5.

[5] Stefan Beyer. What is go ethereum? a detailed guide. Dec 14, 2019. URL https://www.mycryptopedia.com/what-is-go-ethereum-a-detailed-guide/.

[6] Stefan Beyer. What is parity ethereum? a detailed guide. Nov 16, 2019. URL https://www.mycryptopedia.com/what-is-parity-ethereum-a-detailed-guide/.

[7] Bitburner. Proof of weak hands (powh) coin hacked, 866 eth stolen. Feb 1, 2018. URL https://steemit.com/cryptocurrency/@bitburner/proof-of-weak-hands-powh-coin-hacked-866-eth-stolen.

[8] James Chen. Ponzi scheme. Feb 10, 2020. URL https://www.investopedia.com/terms/p/ponzischeme.asp.

[9] Bram Cohen. The bittorrent protocol specification. Feb 4, 2017. URL https://www.bittorrent.org/beps/bep_0003.html.

[10] ConsenSys. A definitive list of ethereum developer tools. Feb 11, 2019. URL https://media.consensys.net/an-definitive-list-of-ethereum-developer-tools-2159ce865974.

[11] Luc Falempin. A closer look at ico smart contracts. Jun 27, 2018. URL https://medium.com/tokeny/a-closer-look-at-ico-smart-contracts-5812aecd782e.

[12] Jacib Johannsen Amrit Akumar Anton Trunov ken Chan Guan Hao Ilya Sergey, Vaivaswath Nagaraj. Safer smart contract programming with scilla. Oct, 2019. URL https://ilyasergey.net/papers/scilla-oopsla19.pdf.

[13] Will Kenton. Hyperledger fabric. Feb 3, 2020. URL https://www.investopedia.com/terms/h/hyperledger-fabric.asp.

[14] Changjia Chen Maofeng Zhang, Wolfgang John. Architecture and download behavior of xunlei: A measurement-based study. June 2010. URL https://www.researchgate.net/publication/229067694_Architecture_and_download_behavior_of_Xunlei_A_measurement-based_study.

[15] Daniel Nyairo. How to use ethereum mist wallet – step-by-step guide. March 21, 2018. URL https://www.chainbits.com/wallets/how-to-use-ethereum-mist-wallet/.

[16] OpenZeppelin. Safemath.sol. Feb 14, 2020. URL https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/math/SafeMath.sol.

[17] Ben Perez. Rubixi.sol. Sep 19, 2018. URL https://github.com/crytic/not-so-smart-contracts/blob/master/wrong_constructor_name/Rubixi_source_code/Rubixi.sol.

[18] Nathan Reiff. What is erc-20 and what does it mean for ethereum? Jun 25, 2019. URL https://www.investopedia.com/news/what-erc20-and-what-does-it-mean-ethereum/.

[19] Nathan Reiff. Decentralized autonomous organization (dao). June 25, 2019. URL https://www.investopedia.com/tech/what-dao/.

[20] Josep Sanjuas. An analysis of a couple ethereum honeypot contracts. May 15, 2018. URL https://medium.com/coinmonks/an-analysis-of-a-couple-ethereum-honeypot-contracts-5c07c95b0a8d.

[21] Martin Holst Swende. Ethereum improvement proposal (eip) 1884. March 28, 2019. URL https://github.com/ethereum/EIPs/blob/master/EIPS/eip-1884.md.

[22] Carla Tardi. Application-specific integrated circuit (asic) bitcoin miner. Sep 2, 2019. URL https://www.investopedia.com/terms/a/asic.asp.

[23] Verify.as. Why dagger-hashimoto for ethereum? Oct 29, 2017. URL https://medium.com/verifyas/why-dagger-hashimoto-for-ethereum-773f0792a689.

[24] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger byzantium version. Oct 20, 2019. URL https://ethereum.github.io/yellowpaper/paper.pdf.

Appendix A

Security Tools

Visualization

1. **Solidity Visual Auditor** - This extension contributes security centric syntax and semantic highlighting, a detailed class outline and advanced Solidity code insights to Visual Studio Code

2. **Sūrya** - Utility tool for smart contract systems, offering a number of visual outputs and information about the contracts' structure. Also supports querying the function call graph.

3. **Solgraph** - Generates a DOT graph that visualizes function control flow of a Solidity contract and highlights potential security vulnerabilities.

4. **EVM Lab** - Rich tool package to interact with the EVM. Includes a VM, Etherchain API, and a trace-viewer.

5. **ethereum-graph-debugger** - A graphical EVM debugger. Displays the entire program control flow graph.

6. **Piet** - Web application helping understand smart contract architectures. Offers graphical representation and inspection of smart contracts as well as a markdown documentation generator.

Static and Dynamic Analysis

1. **MythX** - MythX is a professional-grade cloud service that uses symbolic analysis and input fuzzing to detect common security bugs and verify the correctness of smart contract code. Using MythX requires an API key from mythx.io.
2. **Mythril** - The Swiss army knife for smart contract security.
3. **Slither** - Static analysis framework with detectors for many common Solidity issues. It has taint and value tracking capabilities and is written in Python.
4. **Contract-Library** - Decompiler and security analysis tool for all deployed contracts.
5. **Echidna** - The only available fuzzer for Ethereum software. Uses property testing to generate malicious inputs that break smart contracts.
6. **Manticore** - Dynamic binary analysis tool with EVM support.
7. **Oyente** - Analyze Ethereum code to find common vulnerabilities, based on this paper.
8. **Securify** - Fully automated online static analyzer for smart contracts, providing a security report based on vulnerability patterns.
9. **SmartCheck** - Static analysis of Solidity source code for security vulnerabilities and best practices.
10. **Octopus** - Security Analysis tool for Blockchain Smart Contracts with support of EVM and (e)WASM.
11. **sFuzz** - Efficient fuzzer inspired from AFL to find common vulnerabilities.

Weakness OSSClassifcation & Test Cases

1. **solidity-coverage** - Code coverage for Solidity testing.

Test Coverage

1. **SWC-registry** - SWC definitions and a large repository of crafted and real-world samples of vulnerable smart contracts.
2. **SWC Pages** - The SWC-registry repo published on Github Pages

Linters

Linters improve code quality by enforcing rules for style and composition, making code easier to read and review.

1. **Solcheck** - A linter for Solidity code written in JS and heavily inspired by eslint.
2. **Solint** - Solidity linting that helps you enforce consistent conventions and avoid errors in your Solidity smart-contracts.
3. **Solium** - Yet another Solidity linting.
4. **Solhint** - A linter for Solidity that provides both Security and Style Guide validations.

Appendix B

Installing the Solidity Compiler

Versioning

Solidity versions follow semantic versioning and in addition to releases, nightly development builds are also made available. The nightly builds are not guaranteed to be working and despite best efforts they might contain undocumented and/or broken changes. Latest release is recommendedd. Package installers below will use the latest release.

Remix

Remix is recommended for small contracts and for quickly learning Solidity.

Access Remix online, you do not need to install anything. If you want to use it without connection to the Internet, go to https://github.com/ethereum/remix-live/tree/gh-pages and download the **.zip** file as explained on that page. Remix is also a convenient option for testing nightly builds without installing multiple Solidity versions.

Further options on this page detail installing commandline Solidity compiler software on your computer. Choose a commandline compiler if you are working on a larger contract or if you require more compilation options.

npm / Node.js

Use npm for a convenient and portable way to install solcjs, a Solidity compiler. The solcjs program has fewer features than the ways to access the compiler described further down this page. The Using the Commandline Compiler documentation assumes you are using the full-featured compiler, solc. The usage of solcjs is documented inside its own repository.

Note: The solc-js project is derived from the C++ solc by using Emscripten which means that both use the same compiler source code. solc-js can be used in JavaScript projects directly (such as Remix). Please refer to the solc-js repository for instructions.

```
npm install -g solc
```

Note

The commandline executable is named solcjs.

The comandline options of solcjs are not compatible with solc and tools (such as geth) expecting the behaviour of solc will not work with solcjs.

Docker

ocker images of Solidity builds are available using the **solc** image from the **ethereum** organisation. Use the **stable** tag for the latest released version, and **nightly** for potentially unstable changes in the develop branch.

The Docker image runs the compiler executable, so you can pass all compiler arguments to it. For example, the command below pulls the stable version of the **solc** image (if you do not have it already), and runs it in a new container, passing the **–help** argument.

```
docker run ethereum/solc:stable --help
```

You can also specify release build versions in the tag, for example, for the 0.5.4 release.

```
docker run ethereum/solc:0.5.4 --help
```

To use the Docker image to compile Solidity files on the host machine mount a local folder for input and output, and specify the contract to compile. For example.

```
docker run -v /local/path:/sources ethereum/solc:stable -o
    /sources/output --abi --bin /sources/Contract.sol
```

You can also use the standard JSON interface (which is recommended when using the compiler with tooling). When using this interface it is not necessary to mount any directories.

```
docker run ethereum/solc:stable
    --standard-json < input.json > output.json
```

Binary Packages

Binary packages of Solidity are available at solidity/releases.

We also have PPAs for Ubuntu, you can get the latest stable version using the following commands:

```
sudo add-apt-repository ppa:ethereum/ethereum

sudo apt-get update

sudo apt-get install solc
```

We are also releasing a snap package, which is installable in all the supported Linux distros. To install the latest stable version of solc:

```
sudo snap install solc
```

If you want to help testing the latest development version of Solidity with the most recent changes, please use the following:

```
sudo snap install solc --edge
```

Note

The **solc** snap uses strict confinement. This is the most secure mode for snap packages but it comes with limitations, like accessing only the files in your **/home** and **/media** directories. For more information, go to Demystifying Snap Confinement.

Arch Linux also has packages, albeit limited to the latest development version:

```
pacman -S solidity
```

We distribute the Solidity compiler through Homebrew as a build-from-source version. Pre-built bottles are currently not supported.

```
brew update
brew upgrade

brew tap ethereum/ethereum
brew install solidity
```

To install the most recent 0.4.x / 0.5.x version of Solidity you can also use **brew install solidity@4** and **brew install solidity@5**, respectively.

If you need a specific version of Solidity you can install a Homebrew formula directly from Github.

View solidity.rb commits on Github.

Follow the history links until you have a raw file link of a specific commit of **solidity.rb**.

Install it using **brew**:

```
# eg. Install 0.4.8
brew install https://raw.githubusercontent.com
    /ethereum/homebrew-ethereum
    /77cce03da9f289e5a3ffe579840d3c5dc0a62717/solidity.rb
```

Gentoo Linux has an Ethereum overlay that contains a solidity package. After the overlay is setup, **solc** can be installed in x86_64 architectures by:

```
emerge dev-lang/solidity
```

Appendix C

Gwei (Ethereum)

Gwei is a denomination of the cryptocurrency ether (ETH), which is used on the Ethereum network. Ethereum is a blockchain platform, like Bitcoin, where users transact with each other to buy and sell goods and services without a middle man or interference from a third party

Gwei is short for gigawei, or 1,000,000,000 wei. Wei, as the smallest (base) unit of ether, is like what cents are to the dollar and satoshi are to bitcoin. As with fiat currencies, like the U.S. dollar or the euro, ether is broken into denominations.

New digital currency denominations are becoming popular to help denote the smaller transactions correctly; these may look like very lengthy fractions in terms of ether but equate to high values when converted to U.S. dollars or another fiat currency. The table below displays the typical units of ether (with Gwei highlighted in yellow). You likely would not use them all, however, because transactions on the Ethereum are mostly denominated in ETH or wei.

Unit	Wei Value	Wei
wei	1 wei	1
Kwei (babbage)	1e3 wei	1.000
Mwei (lovelace)	1e6	1.000.000
Gwei (shannon)	1e9 wei	1.000.000.000
microether (szabo)	1e12	1.000.000.000.000
milliether (finney)	1e15 wei	1.000.000.000.000
ether	1e18 wei	1.000.000.000.000.000

Appendix D

Acronyms

P2P Peer to Peer

C2S Client to Server

EOA Externally Owned Account

EVM Ethereum Virtual Machine

ECDSA Elliptic Curve Digital Signature Algorithm

ROM Read Only Memory

DAG Dagger Hashimoto

ASIC Application Specific Sntegrated Circuit

POW Proof of Work

GPU Graphical Processing Unit

POS Proof of Stake

FFG Friendly Finality Gadget

CBC Correct by Construction

API Application Programming Interface

ABI Application Binary Interface

JSON JavaScript Object Notation

DOS Denial of Service

DApps Decentralized Applications

DAO Decentralized Autonomous Organization

PoWHC Proof of Weak Hands Coin

ETC Ethereum Classic

CPU Central Processing Unit

PRNG Pseudo Random Number Generator

ICO Initial Coin Offerings