Master's thesis

Master's Programme in Computer Science

Attacks on Smart Contracts

Otto Porkka

June 7, 2022

Faculty of Science
University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Koulutusohjelma — Utbildningsprogram — Study programme Master's Programme in Computer Science

Tekijä — Författare — Author

Otto Porkka

Työn nimi — Arbetets titel — Title

Attacks on Smart Contracts

Ohjaajat — Handledare — Supervisors

Prof. V. Niemi

Työn laji — Arbetets art — Level Master's thesis	Aika — Datum — Month and year June 7, 2022	Sivumäärä — Sidoantal — Number of pages 48 pages

Tiivistelmä — Referat — Abstract

Blockchain technologies and cryptocurrencies have gained massive popularity in the past few years. Smart contracts extend the utility of these distributed ledgers to distributed state machines, where anyone can store and run code and then mutually agree on the next state. This opens up a whole new world of possibilities, but also many new security challenges.

In this thesis we give an up-to-date survey on smart contract security issues. First we give a brief introduction to blockchains and smart contracts and explain the most common attack types and some mitigations against them. Then we sum up and analyse our findings.

We find out that many of the attacks could be avoided or at least severely mitigated if the coders followed good coding practices and used design patterns that are proven to be good. Another finding is that changing the underlying blockchain technology to counter the issues is usually not the best way, as it is hard and troublesome to do and might restrict the usability of contracts too much. Lastly, we find out that many new automated tools for security are being developed and used, which indicates movement towards more conventional coding where automated tools like scanners and analysers are being used to cover a large set of security issues.

ACM Computing Classification System (CCS)
Security and Privacy $\rightarrow$ Systems Security $\rightarrow$ Distributed systems security

Avainsanat — Nyckelord — Keywords

smart contracts, blockchains, cryptocurrencies

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Networking study track

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Matemaattis-luonnontieteellinen tiedekunta	Koulutusohjelma — Utbildningsprogram — Study programme Tietojenkäsittelytieteen maisteriohjelma

Tekijä — Författare — Author

Otto Porkka

Työn nimi — Arbetets titel — Title

Älysopimusten hyökkäykset

Ohjaajat — Handledare — Supervisors

Prof. V. Niemi

Työn laji — Arbetets art — Level Maisterintutkielma	Aika — Datum — Month and year 7.6.2022	Sivumäärä — Sidoantal — Number of pages 48 sivua

Tiivistelmä — Referat — Abstract

Lohkoketjuteknologiat ja kryptovaluutat ovat olleet murroksessa viime vuosina. Kryptovaluuttojen alkuperäinen ajatus pelkästään hajautetusta ja avoimesta virtuaalivaluutasta on laajentunut huomattavasti älysopimusten myötä, sillä älysopimukset sallivat valuuttasiirtojen lisäksi myös koodin tallentamisen ja ajamisen hajautetusti lohkoketjussa. Nykyään voidaankin puhua jo paremminkin teknologia-alustoista kuin vain kryptovaluutoista. Älysopimukset tuovat luonnollisesti mukanaan myös omat tietoturvaongelmansa.

Tässä tutkielmassa annamme ajantasaisen katsauksen tämänhetkisistä yleisimmistä älysopimusten tietoturvaongelmista. Lohkoketjuteknologiat ja älysopimukset esitellään ensin lyhyesti, jonka jälleen käydään läpi yleisimmät älysopimusten hyökkäystyypit ja niiden vastakeinoja. Sen jälkeen tehdyt havainnot ja löydökset analysoidaan ja käydään läpi.

Havainnoista selviää, että suurin osa esitellyistä hyökkäyksistä pystyttäisiin estämään tai ainakin osittain torjumaan olemalla huolellisempi älysopimuksia kehitettäessä. Hyvät kehityskäytänteet ja hyväksi todetut kehitysmallit auttavat tässä huomattavasti. Osa ongelmista pystyttäisiin välttämään myös tekemällä muutoksia itse lohkoketjuteknologiaan, mutta tämä on usein liian raskas ja työläs tapa torjua ongelmia joihin löytyy usein helpompiakin keinoja. Havainnoista selviää myös, että automaatiotyökaluja käytetään yhä enenevässä määrin torjumaan ja havaitsemaan älysopimusten tietoturva-aukkoja.

ACM Computing Classification System (CCS)
Security and Privacy → Systems Security → Distributed systems security

Avainsanat — Nyckelord — Keywords

älysopimukset, lohkoketjut, kryptovaluutat

Säilytyspaikka — Förvaringsställe — Where deposited

Helsingin yliopiston kirjasto

Muita tietoja — övriga uppgifter — Additional information

Tietoverkkojen opintosuunta

Contents

1 Introduction

Blockchain technology and cryptocurrencies have been extremely popular in the past decade and there is no end in sight. What started as a prototype of unregulated digital money in the form of Bitcoin, has become a multi-trillion dollar industry. Using blockchain as a distributed and immutable database for storing value transactions proved to be truly revolutionary. With blockchain technologies, anyone can now send, receive and verify value transactions over the internet, completely without trusted third parties like banks or governments.

Smart contracts continue this trend by expanding the utility of blockchains beyond simply being backbone for online currencies. By allowing deployment and execution of code in the blockchain, smart contracts essentially turn cryptocurrency protocols into big distributed state machines where anyone can verify and execute deployed code in the same manner as anyone can verify and execute cryptocurrency transactions. Every call and execution of a smart contract gets recorded in the blockchain along with the new state of the smart contract. This new utility brings up a lot of interesting use cases for blockchains and cryptocurrency protocols. Therefore it is no surprise that it also brings up a lot of new security challenges.

Smart contract security is particularly crucial as the immutability of blockchain makes patching vulnerabilities extra hard and a lot of money is usually involved in the form of cryptocurrencies. Smart contracts are also juicy targets for malicious users as the code is readable by anyone and anonymity of the blockchains makes it hard to get caught. Immutability also makes it practically impossible for victims to revert the situation if the attack is successful.

There have been some previous research on smart contract security, e.g. [3, 29, 31, 43], and a lot of research on smart contract utility, e.g. [2, 10, 11, 52, 53]. But as a brand new and constantly changing field, the need for better coverage and understanding of smart contract security issues is only increasing. The major portion of smart contract research and innovation also happens outside of scientific field in the online communities, so to get the up-to-date view on smart contract security, combining the cutting edge open source knowledge and previous scientific work is needed.

This is exactly what is the goal of this thesis, i.e., to provide an up-to-date survey on smart contract security issues. To achieve this goal, the currently most common attack types and their related vulnerabilities are explained and some mitigations against them are presented. Then the overall state of smart contract security is analysed based on the summary of the discussed issues. The smart contract security is defined in this thesis to include technology level security issues as well as issues in the smart contract code itself. Technology level issues must be addressed because the targets of technology level exploits are practically always smart contracts and many of the attacks are also closely intertwined with how the blockchain technology behaves.

This review and analysis reveals that security is of the utmost importance when designing smart contracts. Many of the attacks could be avoided or at least severely mitigated if the developers followed good coding practices and used design patterns that are proven to be good. On the other hand, changing the underlying blockchain technology to counter the issues is not an optimal counter measure, as it is hard and troublesome and might restrict the usability of contracts too much. Some technology level behaviour can even be desired, even if it allows misuse when combined with badly designed smart contracts. Lastly, new automated tools for security are being developed and emerging all the time. This indicates movement towards more conventional coding where automated tools like scanners and analysers are being used to cover a large set of security issues.

The structure of the rest of this thesis is as follows. First, in the chapter 2, the basics of the blockchain technologies, cryptocurrencies and smart contracts are given. Section 2.1 defines blockchains, section 2.2 explains the basics of cryptocurrencies and section 2.3 defines smart contracts. Section 2.4 explains the Ethereum cryptocurrency protocol in a bit more detail. This is needed, because Ethereum is generally considered to be the first true smart contract capable cryptocurrency protocol. It is also by far the biggest and most used one, and most of the explained attacks are against Ethereum smart contracts. In the chapter 3, attacks and vulnerabilities themselves are explained. Attack types are not in any specific order, but as a more general type, *Front-Running* in section 3.3 is split into sub-cases of *Transaction Ordering Dependence*, *Block Stuffing* and *Block Reorganization Attacks*. After each separate attack type, some possible mitigations against them are also presented. In the chapter 4, all the covered attack types and their mitigations are summed up and analysed to give a better understanding of the big picture of the current state of smart contract security. In the section 4.1, some thoughts about the possible future of smart contract security and smart contract research in general are given. Finally, in chapter 5, a few final words are given to sum up and conclude this thesis.

2 Background on Smart Contracts

Before diving into smart contracts and their attack vectors, one must understand what smart contracts and cryptocurrencies are and what is meant by blockchain technology they are built upon.

First, the more abstract view of blockchains and cryptocurrencies is given. Mainly what is the basic idea behind them. Then the definition of smart contracts is specified. Lastly, the Ethereum protocol is discussed in a bit more detail to give a more concrete example. Ethereum is chosen because it is the first protocol to implement smart contracts, and at the time of writing this, Ethereum has a huge dominance over other protocols with smart contract capabilities.

2.1 Blockchain technology

The basic idea of blockchains is not very new. On a very abstract level *blockchain* is a distributed list of records where a set of nodes contribute to maintain and update it. An idea that dates back to as early as the 1970s. Another key characteristic of blockchains is that they are immutable, meaning once the information is in the chain, no one is able to change it or erase it. [44] The property of immutability proves to be extremely useful when blockchains are utilized in cryptocurrencies, from which they are the most known of. The way that immutability is achieved is by chaining blocks of information via cryptographic hash functions, hence the name "blockchain". The next block of data always uses the hash of the block right before it, making it impossible to modify information later on. This idea has been discussed already in 1979 by Ralph Merkle with *Merkle hash trees* and implemented in 1990 by Sutart Haber and W. Scott Stornetta in so called time-stamp documents. [44]

The term *blockchain* or *blockchain technology* itself can have many definitions. Strictly speaking it only means immutable blocks of information that are linked together in an ordered chain using cryptography. It does not really matter if it is distributed or centralized or if users must be trusted or not. This definition can be expanded to the more common use case mentioned above where the chain is maintained by a peer-to-peer network of nodes, as is the case with cryptocurrencies. This is also the definition that IBM uses, for example. They state that blockchain is “shared, immutable ledger”, where the term “shared” means that it is distributed and the term “ledger” refers to the common use case of recording value transactions. [25]

2.2 Cryptocurrencies

The broadest definition for *cryptocurrency* is that it is a type of digital money or digital currency. As with any other currency, its main function is to act as a medium of exchange. What makes it different from other digital currencies is that it builds on the blockchain and relies on some sort of cryptographic trust to reduce the involvement of third parties or intermediaries. [39]

The common way of handling payments over the internet rely on these trusted third parties, which are usually banks and other financial institutions. Since they must be able to reverse transactions in case of disputes, their systems cannot be truly immutable. This possibility of reversal also spreads the need for trust. One must always have enough information of other party to gain trust before transactions and with third parties the extra costs are also present. In the case of cryptocurrencies, the trust in the third party is replaced with the trust in the consensus of the nodes and the immutability of the blockchain [39]. The consensus itself is the trusted chronological order of all transactions, which is considered final once in the blockchain and therefore cannot be reversed or modified. This allows users of the given cryptocurrency to send and receive value over the network completely without intermediaries or extra information needed. For example, Satoshi Nakamoto (alias) uses this as the base motivation for Bitcoin in his famous paper “Bitcoin: A Peer-to-Peer Electronic Cash System” [36]. This “lack of trust” in cryptocurrencies is why they are often referred as “trustless”, although it is a bit misleading since in reality the trust is in the consensus and honesty of the co-operating nodes.

A *cryptocurrency system* is a system built on blockchain technology that issues tokens and uses them as a medium of exchange, and the token itself can be thought as the actual “cryptocurrency” [39]. However the system itself, or the “protocol”, which is how they are usually referred to, can also be referred to as a cryptocurrency. Protocols usually make the distinction between the system and the token by naming them differently. The most known examples are Bitcoin (the protocol) and bitcoin (the cryptocurrency), and respectively Ethereum (the protocol) and ether (the cryptocurrency). It has also become really vague as to what actually counts as cryptocurrency as the modern protocols often differ greatly on their architecture and governance and they have become much more than simple mediums of exchange. Ingolf Pernice and Brett Scott try to find the definition for the term in their paper [39] and they come to a conclusion that “cryptocurrency systems are unified by being intended to host a general or limited-purpose medium-of-exchange, a cryptocurrency, using infrastructure that replaces trust in institutions by cryptography to varying degrees.” So in the end, the defining properties of cryptocurrencies are their “trustlessness” and their ability to act as a medium of exchange. In the terms of this paper we however avoid using the term “cryptocurrency”, if possible, due to its vague nature and instead prefer the terms “protocol” and “token” describing the cryptocurrency system and its respective cryptocurrency.

As the topic of this thesis addresses smart contracts and their attack vectors, it is useful to be a bit more specific on what kind of protocols we are talking about. Even if cryptocurrency protocols can also be implemented by institutions or states, we focus solely on open and decentralized protocols. This means that all the code is open source, as it would be hard to discuss attacks if you cannot see the actual code. It also means that no centralized authority is behind issuance of the tokens or the governance of the protocol.

2.2.1 Mining and Consensus Algorithms

As was the case with blockchains, the idea of cryptocurrencies also dates back to the last millennium. In 1998 Wei Dai proposed the idea of creating value for digital money through solving computational puzzles and in 2005 Hal Finney came close to modern cryptocurrencies with his Reusable Proofs of Work. They both relied on the idea that issuance and scarcity, or in other words, the value of the currency, can be implemented by awarding tokens to users that showed a proof of solving some hard computational problem. This process of solving computational puzzles for tokens is usually referred to as “mining” and we will also call it that in this paper. Generation of tokens, in any context, is called “minting”. After showing the proof, the solver then has some amount of tokens that they can send to some other user by signing the transaction with their private key. Public keys of users act as their receive addresses. [7]

However, all these prior prototypes and proposals lacked the consensus algorithm or relied on some form of third party for updating and maintaining the ledger. It was not until Satoshi Nakamoto and Bitcoin that the true breakthrough happened. Bitcoin was the first truly decentralized protocol. It achieved this feat by using blockchain as underlying data structure and combining issuing new tokens and maintaining the ledger to the same algorithm. [7]

How the algorithm works in practice is that every time some miner node successfully solves the computational puzzle, it is awarded some bitcoin and the solution is used as hash of the next block in the blockchain. Miner also inputs transactions it has received from the network in the block, thus participating in the updating the ledger. The consensus itself is achieved by only accepting the longest chain as the valid one. This means that in case there are multiple valid blocks mined at the same time or if some malicious node or group of nodes try to modify transaction history, they must recalculate all of the hashes of the previous blocks starting from the block where the difference happened. This would be highly impractical as the attacker or fork of the chain would need by average over half of the computing power of the network to outrace the longest chain. For the nodes it is therefore better to be honest and race for the next hash and reward than to fight the whole network. [36] This type of attack where the attacker controls the majority of the network is called *51%-attack*.

This type of consensus algorithm where nodes need computing power to participate, referred as *Proof of Work* (PoW), has an additional and really useful benefit of mitigating so-called sybil attacks. Sybil attack is a type of attack where the attacking party tries to take over the network by creating multiple entities. In the case of cryptocurrencies this means that the attacking party would create or imitate so many nodes that they would form the majority and be able to dictate the consensus of the network. They would then be able to, for example, double spend tokens by reversing the transaction by choosing another fork. However, since the power the user has in the network is not tied to the amount of nodes but the amount of CPU power, simply having many nodes is not enough. [36]

The obvious problem of energy consumption with Proof of Work -type of algorithms should also be addressed. For example, a CoinDesk article from the August 2021 estimated Bitcoin network energy consumption at around 80 TWh (terawatt hours) annually [19]. This places Bitcoin among the top 50 countries per energy consumption, even while being on the lower end of estimates which usually are anything between 50 to 150 TWh. This problem has been obvious right from the start when it became clear Bitcoin would become popular, and for this reason there has been active research and debate for all this time to replace PoW with something more efficient. Multiple candidates and their various combinations have been proposed, one of the most promising being so called *Proof of Stake* (PoS). Basic idea behind Proof of Stake is to replace the power-intensive CPU mining with the "staked" value, which in theory gives the node right to generate portion of the new blocks based on the amount of tokens it has locked in the protocol [4]. However, these candidates escape the topic of this thesis and are just something that is good to acknowledge.

2.3 Smart Contracts

Classical protocols like Bitcoin only focus on being a medium of exchange. They issue tokens and provide a peer-to-peer network to handle transactions and maintain the state of the system. The attributes of the underlying blockchain technology however provide many other interesting opportunities. Security, anonymity and data integrity all without trusted third parties can also be a game changer in many other use cases. Blockchains are already used in IoT, smart property, digital content distribution, and of course, smart contracts, to mention few examples [52]. This revolution of blockchains, so to speak, is often referred on the field as web 3.0 or Web3.

Extending the underlying blockchain to more general use cases was also the motivation behind Ethereum, which is generally seen as the foundation of smart contracts and web 3.0. Although Bitcoin provided a weak version of smart contracts via simple scripts, it was not suitable for more general use cases and had some limitations. The Bitcoin version of smart contract is a simple script that owns some amount of tokens. Then when some other user triggers the script by sending a transaction to its public address and its conditions are fulfilled, the tokens are sent to some other address. Scripting language itself is still not Turing-complete as it lacks loops, for example. Bitcoin protocol also allows only spending all of the tokens or none at all which means there can be no internal state and contracts can only be used once.[*] Ethereum fixed these limitations by providing its own Turing-complete scripting language and splitting user accounts to externally owned accounts and contract accounts. Contract accounts have contract code which dictates its behaviour and conditions, and storage which is used to hold information about contracts state. [7]

Now a definition for *smart contract* can be given. It is an autonomous account living in the blockchain with the code that dictates its behaviour. It is typically invoked by sending a transaction to the address of the account, and if accepted by the network, the contract code is run. The state of the blockchain, invoking transaction itself and its payload are used as input. Output of the execution are possible transactions and changes in the internal state

[*]A recent Bitcoin update launched in November 2021 called Taproot just improved Bitcoins smart contract capabilities, but specifics about it escape the topic of this paper. [32]

of the contract if present. By running the code on the whole network, it is made sure that the output is agreed on and that the contract is being honored. [31]

2.4 Ethereum

Now that the basics of cryptocurrencies and smart contracts are explained, it is useful to give a more concrete example on how things work in practice. Ethereum, which has huge dominance over smart contract space, is the best candidate for this.* It is good to mention here that Ethereum, like almost all the other protocols, is under active development and the small details may be subject to change. Some of these details are however essential in the way some of the attacks work so they should be discussed. Whole chapter 2.4 is based on Ethereum documentation unless explicitly indicated otherwise. [17]

2.4.1 EVM and Native Token

As discussed before, blockchains are essentially distributed databases which state the majority of the miner nodes in the network mutually agree on. By adding Turing-complete smart contracts in the blockchain, Ethereum makes the whole network one big distributed state machine. This state machine is commonly referred to as *Ethereum Virtual Machine* (EVM). Every user of the network can request EVM to execute arbitrary code via invoking its respective smart contract by sending a transaction request to its address. Each miner then verifies and validates the transaction and executes the smart contract code, and again mutually agree on the next state of the EVM. As the whole state of EVM, including the internal state of smart contracts, and all the transactions that have ever happened are stored in the blockchain, no one can tamper with them later on. This also provides a way for anybody anywhere and any time to check if execution or transaction really took place.

It would be highly impractical to allow anyone to execute their code on the EVM for as long as they wanted or store as much data on the chain as they wanted, so some limitations are needed. The miners who verify and execute transactions and execute smart contract code also need some kind of incentive to do so. This is where protocols native token,

*According to cryptocurrency tracking site *coingecko.com* [12], on 26.5.2022 the market cap of Ethereum was 225 billion whereas total market cap of all smart contract cryptocurrencies was 381 billion. This would place Ethereum's share at around 59% based on native token market cap only. Share in smart contract usage can be expected to be even higher since the most valuable and most used projects are all based on Ethereum.

ether (ETH), steps in. ETH is minted the same way all the other native tokens are usually minted, by allowing miners to reward themselves with fresh ETH every time they successfully generate a new block. Native token refers to the primary token of the chain which is used as storage of value and medium of exchange. On top of rewarding block generation, ETH is also used as a fee on every operation that happens on blockchain and tipping the miners for executing the wanted operation. These required fees are referred to as "gas" which can be thought of as a usage fee everyone must pay in order to use the EVM, the same way gas is required to operate a vehicle. Pricing of operations also provides a market for EVM computation since these fees change based on demand on the network and higher tipping fees are preferred by miners.

Gas fees are in principle paid in ETH, but all prices and value transactions in the code are actually integers. To allow small fractions, smaller units of "wei" and "gwei" are used. Wei is the smallest possible unit on Ethereum and it is 10^{-18} ETH. In other words, 1 ETH = 1 000 000 000 000 000 000 wei. Gwei is short for giga-wei, or billion weis. 1 ETH is then 1 000 000 000 gweis.

2.4.2 Accounts

Accounts are the main users of the network. Although miners, or nodes, can also be thought of as users, their function is different as their purpose is only to keep the system running. In Ethereum there are two types of accounts, externally-owned accounts and contracts. Both types of accounts can receive, hold and send ETH and tokens, but there are also some crucial differences.

External accounts are the ones that humans can control. They can be controlled by anyone that has the private keys to sign transactions on behalf of the account. Public key is the account address, against which other participants of the network can check the validity of the transaction signatures and also send tokens to. External accounts are the only ones that can initiate transactions. This means they can request transactions from themselves to other external accounts or to smart contract accounts. Transactions between external accounts can however only be ETH or token transfers. Generating external accounts is free, since in practice anyone can generate public and private key pairs and transfer tokens to that public key.

Contract accounts or broadly speaking, smart contracts, have an expanded utility but they cannot be changed or controlled by anyone once deployed on the chain. They can be thought of as autonomous accounts living on blockchain. Key difference from external accounts is that contract accounts are deployed with the code that dictates their behaviour. They also have storage which can be utilized for storing their internal state between executions. As they are autonomous and not controlled, they cannot execute unless explicitly invoked. But once invoked by sending a transaction to them, the code involved with them executes, and they can in practice do about anything. This includes new transactions, creating new contracts, and anything else the code tells them to do. Note that although contract accounts cannot initiate transactions by themselves, they can indeed create and execute new ones once invoked. Deploying (creating) a contract account has a cost since now the storage of the network is involved. Smart contracts are discussed a bit deeper in the section 2.4.4 as their security is the main concern of this paper.

In Ethereum, all accounts have four main fields. *Nonce* is a counter that indicates the number of transactions sent from the account. It ensures all the transactions are processed only once. *Balance*, as one might guess, is the amount of ETH that the account is currently holding. It is denominated in wei. Last two fields are used only by contract accounts. *storageRoot*, or *storage hash*, is a hash identifying the root node of the storage tree structure that holds the contract's internal state. Lastly, *codeHash* points to the code of the contract in the state database that contains the code that is executed when the contract is invoked.

Another commonly used term closely related to accounts is "wallet" or "crypto wallet". A wallet however is only the key-pair associated with the user-owned accounts and not the account itself. It can also refer to a piece of software that holds the private keys and handles the signing and communication with the network on behalf of the user.

2.4.3 Transactions and Gas

Transactions are the bread and butter of every cryptocurrency protocol. They are used to update the underlying ledger. In the case of Ethereum and similar protocols, this also means they are used to update the distributed state machine. In their simplest definition they are cryptographically signed requests from accounts to the network, the most common use case being transferring ETH from one account to another. Ethereum documentation is a bit ambiguous on the definition as it is first said that a transaction is "an action initiated by an externally-owned account" and later on that also contract accounts "can send transactions over the network". For the sake of simplicity, in this thesis a transaction is defined as the request to change the state of EVM.

Before anything happens, the transaction must be formed and signed by an account. Ethereum transactions contain seven fields, the most important ones being *recipient*, *signature* and *value*. *Recipient* is the address of the receiver of a transaction, which can be externally-owned or a contract. *Value* is the amount of ETH which the sender wants to send, denominated in wei. And *signature* is the proof that the one who sends the transaction request actually owns the private keys of the sending account. An optional field *data* is used to include arbitrary data in the transaction that is stored in the blockchain with the transaction.

Last three fields, *gasLimit*, *maxPriorityFeePerGas* and *maxFeePerGas* are used for pricing in the computation and storage requirements for executing the transaction. *GasLimit* is the maximum amount of gas units that the sender is willing to pay for a transaction. These gas units refer to the computational steps required in execution of transaction as well as data units stored in the blockchain [30]. *MaxPriorityFeePerGas* is an extra tip paid to the miner per gas unit consumed. Higher tip naturally gives a transaction higher priority as miners prefer to include transactions with higher tips. This is also the only fee miner gets apart from reward for successfully mining a block. Lastly, *maxFeePerGas* can be used to limit the maximum price that sender is willing to pay for gas. This is the sum of maxPriorityFeePerGas and *baseFeePerGas*, which is per block value set by the protocol based on the demand of the network.

With these last three variables, the sender can limit the maximum number of steps for computation as well as maximum price paid per gas unit. This protects the sender from unexpected costs. Difference between limits set and actual gas used is returned to the sender. In case gas runs out when executing the transaction, the miner keeps the tip but everything else is returned to the sender. State is also not changed. As mentioned when the EVM and native token were discussed in (section 2.4.1), these gas fees are also needed to prevent malicious actors from consuming network resources to prevent legit usage.

After the transaction object is formed, it then needs to be sent and broadcast to the miner network. Transaction gets an identifying hash and it is included in the pool of all pending transactions. To get validated and executed, a miner must include it in the mined block. Only after the transaction is in the block and in the blockchain, can it be considered successful. There is still a small possibility even after this that some fork of the chain becomes the longest one and invalidates the transaction. However as the new blocks get added, the chance for this to happen rapidly decreases, thus “cementing” the transaction in the blockchain. To indicate this, transactions have a number of “confirmations” which is the count of blocks mined after the one the transaction is in.

2.4.4 Ethereum Smart Contracts

As stated before, a smart contract can be thought of as an autonomous account living in the blockchain with its code and state. The code deployed with it dictates its behaviour and it cannot be changed. It has to be deterministic since all the nodes must end up with the same result given the same input. Otherwise consensus would be broken. It also has to be permissionless, meaning anyone can call them by sending a transaction to the contract's address. Anyone can also deploy smart contracts as long as they have enough ETH to pay for it.

One way to think of smart contracts is as open APIs. Web applications can then build on top of them and use them as backends. This type of applications are called *decentralized applications*, or *dapps*. Dapps aim to benefit from the same things as cryptocurrency protocols in general. Security, privacy and data integrity without need for trusted third parties are useful traits in almost any application. Immutability and robustness of cryptocurrencies is still not without drawbacks. Immutability makes it hard to maintain smart contract code. Scaling is hard since executing on the EVM is very resource expensive. Frontend can still be more or less centralized, thus somewhat countering the blockchain mantra of decentralization. On top of all, executing is really slow since every call needs to be mined into the blockchain.

Openness of the smart contracts makes them still really attractive for many use cases since it is easy to reuse and combine them. Contracts can even deploy other contracts. This formation of intertwined smart contract systems has proven to be useful in many fields, the most prominent being *decentralized finance* (DeFi), where contract based systems are challenging financial institutions like banks and exchanges.

As the code and transactions are run on miner nodes, the EVM needs to provide an abstraction between executing code and executing machine. The EVM uses a set of opcode instructions to do this. Smart contract code, when deployed, has to be compiled to EVM bytecode representing these instructions. EVM itself executes as a stack machine with a depth of 1024 items and each item as 256-bit word. Stack machine simply means that all the computation is performed in the stack data structure instead of separate registers.

At the time of writing this, there are two main programming languages used for Ethereum smart contracts, Solidity and Vyper. From these two, Solidity is the more robust one; object-oriented, statically typed, curly-bracket language which supports inheritance, libraries and user-defined types. Vyper on the other hand is a python-like, strongly typed language with stripped capabilities compared to Solidity. Its aim is to make contracts more secure and easier to audit via simplicity.

Contracts naturally have to preserve some information during the executions as well as between them. For this reason there is *memory* for runtime data and *storage* for persistent data. Any contract data must be assigned to one of these locations.

Storage is simply a collection of contract's state variables that get permanently saved on the blockchain. They have to be declared with a type so that storage needs can be calculated. Storage itself is a key-value store which maps 256-bit words to 256-bit words [45]. Head of this structure is pointed at by the storageRoot field of the contract account.

Memory variables are used for runtime data. As they are not saved on the blockchain, they are much cheaper to use. Runtime memory is allocated in 256-bit wide sectors which can be accessed at byte level or as whole sectors. Gas is paid on sector basis every time read or write expands to a new sector, or in other words, when a new 256-bit wide sector is allocated.

Smart contract functions can be categorized in a few different ways. First of all, they can be split into internal or external functions. Internal functions can only be called directly in the contract code and they are basically simple jumps inside the EVM. External function calls on the other hand will always need a transaction, which is often referred as a "message call" if the sender is a contract account. User controlled external accounts can only call external functions by sending transactions to them. Functions can also be public or private. This means the same as in traditional programming. Public functions can be called from anywhere and private functions only from the same contract. Take note that public and private distinction is a visibility definition, but internal and external distinction only matters if the program is making a new EVM call. Lastly, functions can be split into view functions and write functions. View functions do not modify the state of the contract. Write functions, as the name suggests, do. There is also a special type of constructor functions, which are only called when a contract is first deployed.

To give a more concrete example of Ethereum smart contracts, a code snippet is in order. Listing 2.1 shows a small contract which demonstrates storage usage. First line tells source code licence, which is important since smart contracts are open source by default. Second line tells the Solidity version to prevent compiling with a possibly breaking version. Contract itself features the state variable called `storedData`, and two public functions. Variable is typed as `uint` to tell the compiler exactly how much storage is needed. From the two functions `set` and `get`, `set` is a “write” function for modifying the variable and `get` is a “view” function for retrieving its current value.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Listing 2.1: Smart contract example for storing and exposing a single variable [45].

3 Smart Contract Attack Types

Before diving into actual attacks and vulnerabilities, some clarification is needed in what is actually meant by smart contract attacks in this thesis. As explained in the chapter 2, smart contracts are pieces of code that are deployed into a blockchain and are executed by the nodes in the cryptocurrency protocol. In a sense smart contracts then lie between all the applications using them and the blockchain technology itself.

This thesis focuses solely on attack types that use vulnerabilities in the smart contract code, leverage the blockchain technology properties, or need both combined. Even if the vulnerabilities in the underlying blockchain technology might not be directly related to the smart contracts, those vulnerabilities are still commonly used to target smart contracts. Mechanics of the blockchains are also relevant to many of the attacks even if the vulnerability itself might lie in the smart contract code, so mechanics of the blockchain technology and its vulnerabilities must be discussed.

Since the applications using smart contracts can in practice be anything at all, attacks leveraging off-chain applications to attack smart contracts are left out from this thesis. This includes attacks like phishing, fake websites or hacked credentials.

Most of the discussed attacks types need both the technology properties and vulnerable smart contract code to be effective. These include *Re-entrancy* in section 3.1, *Timestamp Dependence* in section 3.2, *Transaction Ordering Dependence* in section 3.3.1, *DoS With Revert* in section 3.5, *Insufficient Gas Griefing* in section 3.6, *Forcibly sending Ether To A Contract* in section 3.7 and *Oracle Manipulation* in section 3.8. Attacks leveraging only the underlying blockchain technology and its properties are *Block Stuffing* in section 3.3.2 and *Block Reorganization Attacks* in section 3.3.3. Finally, the only attack type that exploits just the contract code is *Integer Overflow and Underflow* discussed in section 3.4.

3.1 Re-entrancy

Re-entrancy is probably the most iconic exploitable smart contract vulnerability. It was the cause of the famous Ethereum attack against a contract named "The DAO", which led to a loss of over 3.6 million ETH. The attack was so severe that the Ethereum community voted to return the state of the blockchain and EVM to the one before the hack, causing the original hacked version of Ethereum to continue as a fork called Ethereum Classic. [34]

Re-entrancy itself means that multiple invocations of a function can run concurrently, in other words, a function can be called again even when it is in the middle of execution. With smart contracts this can happen when a smart contract calls another smart contract in its code and waits for this new call to finish before continuing execution. The receiving contract of this call can then execute its own, and possibly malicious, code and recursively call the original contract again. However the original contract can now be in exploitable state because it is in the middle of the original execution. [31]

In the EVM, the transfer of control to the malicious contract is possible because of *fallback* function. Fallback function is executed when a contract is called but no function matches the call, or when a contract receives ETH transfer [41]. The latter is the more usual case since value transfers are much more common than calling function that does not exists, which is rarely a wanted behaviour.

```
1  contract Victim {
2      mapping(address => unit) public balances;
3
4      function deposit() public payable {
5          balances[msg.sender] += msg.value
6      }
7
8      function withdraw(uint amount) public {
9          require(balances[msg.sender] >= amount);
10         (bool sent, ) = msg.sender.call{value: _amount}("");
11         balances[msg.sender] -= amount;
12     }
13 }
```

Listing 3.1: Victim smart contract.

To understand how the re-entrancy attack works in practice, an example of an exploitable smart contract can be seen in the listing 3.1. Contract stores ETH for its users in the key-value table called `balances`, seen on line 2. Every user of the contract has their address mapped to the sum they have stored previously by calling the `deposit` function on line 4. Stored ETH is added to the balance of the contract account itself. The re-entrancy vulnerability is in the function `withdraw` on line 8. On line 10, the contract transfers funds to the caller of the function if it had enough balance in the balance table. This will trigger the fallback function of the caller if it is a smart contract. Now the contract will wait for this call to return before actually subtracting value in the balance table on line 11.

If this contract would be deployed and had users store funds in it, it would be easily exploitable by crafting a malicious smart contract. Such a contract can be seen in the listing 3.2 and the exploit itself is executed by calling the `attack` function on line 10. To start off, the attacker needs some balance on the balance table. This is done by storing some value on the victim contract by simply calling its `deposit` function. The vulnerable `withdraw` function is called after this.

```
 1 contract Attacker {
 2     Victim public victim;
 3 
 4     fallback() external payable {
 5         if (victim.balance >= 1 ether) {
 6             victim.withdraw(1 ether);
 7         }
 8     }
 9 
10     function attack() external payable {
11         require (msg.value >= 1 ether);
12         victim.deposit{value: 1 ether}();
13         victim.withdraw(1 ether);
14     }
15 }
```

Listing 3.2: Attacker smart contract.

Now the victim contract checks the balance from the table and transfers funds back, triggering the `fallback` function on line 4. At this point the execution of the victim contract is halted and balance is not yet subtracted. Attack contract abuses this halted state by calling the `withdraw` function again and since the balance still checks out, funds are unintentionally transferred again. This triggers yet another `fallback` function execution, which calls `withdraw` function again, et cetera. This cycle repeats recursively until balance check on the line 5 informs that the attacker contract has drained the victim contract from all its funds. After this the whole call stack unwinds and the balance is finally subtracted.

Above example is called cross-function re-entrancy, since originally called function and repeatedly called function are different but share internal state. Variation where only one function of victim contract is used is respectively called a single function re-entrancy [13]. From these two, the cross-function re-entrancy vulnerabilities are obviously much harder to detect since they involve more components.

Mitigations: Better coding practices. To mitigate re-entrancy exploitation, there are few things that can be done. First thing is that developers should understand how re-entrancy works and pay more attention to the code they are writing. A great practice for developers, listed in Ethereum documentation, is to simply design contracts so that they "neither send ETH nor call untrusted contracts". By doing this, the control of execution stays within the trusted contracts and malicious code never even has a chance to recursively call back. If sending ETH or calling untrusted contracts must still be done, then it is really up to the developer to check the control flow so that possibility for exploitation is mitigated. [17] For example, to fix the code in listing 3.1, one could subtract balance first and only send ETH after this. This solution also follows what blockchain company ConsenSys recommends for re-entrancy prevention [13]. They state that all internal work, including state changes, should be completed prior to calling any external function, including sending ETH. Other possible way to fix the example would be to check that the caller is never a smart contract [17]. This would however limit the usefulness of the contract.

Mitigations: Technology changes. Smart contract technologies seem to not be inherently able to prevent re-entrancy exploits, as they are designed to be as versatile as possible. A simple solution would be to allow locking of the contract or function for the duration of the execution so it cannot be called again. However, this can already be done by coding and would be hard to implement on the technology level unless the blockchain is brand new.

Mitigations: Tools. One prominent way to prevent re-entrancy bugs, that is also widely used for conventional coding, is automated tools. Current tools range from static analysis, to dynamic and symbolic analyzers and fuzzers [43]. Even deep learning have been proposed and studied as one solution in catching smart contract bugs and vulnerabilities [41]. Automated tools have already found their place as an irreplaceable help in writing secure smart contracts.

3.2 Timestamp Dependence

When a miner proposes a new block to be added into the blockchain, it must assign a timestamp to it. The timestamp of the block will have to fulfil two conditions to be accepted by the network. It can never be earlier than the timestamp of the previous block. [17] It should also not be too far in the future. For example, popular Ethereum implementations Geth and Parity reject timestamps that are more than 15 seconds after the timestamp of the previous block [13]. Within these limits the timestamp is still fully controlled by the node mining the given block.

Allowing an arbitrary node to decide the block timestamp may not sound that critical in general, but problems can arise when the timestamp is used in smart contract code. In Ethereum, the timestamp can be accessed via global variable called "block.timestamp" [17]. A common example where the usage of this variable might be problematic is when it is used as a seed for a random number. Now the node that mined the block can pick any timestamp in the valid time window and try to find one that favours the outcome it wants. Or even worse, the node can pick a timestamp that gives the exact outcome it wants. [43] Block timestamps should therefore never be considered truly random or exact, and critical operations should never trust on them [31].

Mitigations: Better coding practices. To promote better coding practices, ConsenSys recommends something called "the 15-second rule" [13]. What this means is that timestamp should only be used in a way that 15 second variation in the timestamp does not affect the outcome. This essentially makes it impossible for a miner to affect the result of the code execution even if it would tamper with the timestamp. Another suggestion is to use block numbers instead of timestamps, since in most cases the block number multiplied by block produce time serves the same purpose as the block timestamp [31]. Note that it is impossible for a node to modify the block number as it is simply a count of all the blocks in the blockchain. However, this solution is susceptible to chain reorganisations and changes in block produce time, so for example Smart Contract Weakness Classification Registry (SWC Registry) do not recommend it [46]. Instead, they recommend developers to take caution when using block values like block number and timestamp, as they are not precise, or to use oracles. Oracles are services which act as interfaces between non-deterministic real world and deterministic on-chain smart contracts [9]. They aim to solve the problem of accessing off-chain data or delivering data off-chain in a deterministic way.

Mitigations: Technology changes. Because timestamp is mainly needed by miners for block validation, one viable solution would be to restrict smart contracts from accessing it altogether [31]. Being able to access timestamp is not essential for smart contracts as there are better ways to obtain random seeds.

3.3 Front-Running

Front-running as a term originates from the financial world. In that context it means trading of an asset by a broker, which tries to benefit from knowing about an upcoming transaction that is going to affect the asset price [35]. Broker can use this knowledge to execute its own transaction first, hence the term “front-running”. For example, if a stock broker gets a large order from a client to buy stock “X”, it can first buy the same stock itself, and then sell right after the client’s buy order is executed, thus making instant profit from a price difference. This works because typically the asset price increases if it is bought a lot.

In a cryptocurrency world, “front-running” has a bit broader definition and can be thought of as an umbrella term for many different activities. However, the main idea remains that some actor cuts in line of other transactions to gain benefit, most of the time based on some vital knowledge about those other transactions [13].

It is good to note at this point that although front-running is highly illegal in traditional finance, with cryptocurrencies it is generally considered acceptable and in some cases even beneficial for the common good. Good examples are *arbitrage* and *sandwich trading*, which are explained in more detail later in section 3.3.1. Arbitrage can ensure that all users get the best and most correct prices for their tokens. On the other hand, sandwich trading makes the price worse for users, and benefits only the front-runner. [17] Because of this, when the term “attack” is used in front-running context, it refers more to some action leveraging front-running than an action with a malicious purpose. Morality of practicing front-running is therefore not dependent on the technical details.

What makes front-running so easy with common cryptocurrency protocols is that anyone can see all the transactions that are requested as they are in transaction pool waiting to be included in a block [17]. Then to actually cut in line, one can offer miners a really high tip to improve the chances that their transaction is run before the others [13]. This is the usual way that front-runners operate. These type of actors are also commonly referred as “searchers”, as they are often bots that hunt for front-running opportunities. Searchers can also front-run other searchers by searching the transaction pool for their font-running attempts and then copy and outbid them. This type of searchers are called *generalized front-runners*. Because of this bidding competition, a big part of the value that searchers make is actually going to miners as tips. And as bidding gets more competitive and searchers become more optimized, they are willing to sacrifice bigger margins just to get their transactions executed. [17]

Another possible way to do front-running is to actually be the miner that mines the block, as the miner has all the power to decide which transactions to include in the generated block and in which order. Extra value generated by miners by doing front-running and reorganizing transactions is called *Miner Extractable Value* or *Maximum Extractable Value* (MEV), and it includes all the value generated from block production apart from block reward and gas fees. Value generated by searcher bots is usually also counted as MEV, since it is gained by front-running and ordering of transactions. [17]

On top of exploiting the ordering of transactions inside the block, miners can also gain value from trying to rewrite the tip of the blockchain and rewind history [16]. This way they can benefit from previous MEV opportunities as well as generate new ones from reorganising the whole chain. Reorganizing of blocks is a serious concern with potential to destabilize entire protocols. Two cases of reorganization attacks, *Time-Bandit Attack* and *Undercutting Attack*, will be discussed section 3.3.3.

Now that the basics of front-running are explained, one could argue that front-running is actually not a smart contract attack, since it is the cryptocurrency protocol itself, together with miners and transaction bidding, that makes cutting in line possible in the first place. However, targets of front-running are in practice always smart contracts, and the vulnerability itself lies in the fact that outcome of contract execution is usually dependent on order of transactions [16]. This is why the vulnerability itself that front-running exploits is called *Transaction Ordering Dependence*. General case of transaction ordering dependence, with common examples is discussed in section 3.3.1 and a more specific attack of *Block Stuffing* in section 3.3.2.

3.3.1 Transaction Ordering Dependence

As the name suggests, *Transaction Ordering Dependence* means there exists some sort of dependency between transactions, and the order they are executed has an effect on the outcome. This creates race conditions, where firstly executed transactions can have benefit over others [46]. In this context the dependence itself should be considered as a vulnerability in the contract code, and front-running activities that try to exploit it are the actual attacks. It is also good to note that practically every smart contract has some potential for transaction ordering dependence, although most cases are benign. [16]

Attacks exploiting transaction ordering dependence can be further split into *displacement* attacks and *insertion* attacks [13]. In displacement attacks all that matters is that the attacker gets to execute first. It does not matter whether other users execute their transactions or not. On the other hand, in insertion attacks, the profit is generated from the other transactions that are run after or before the front-runner.

A common target for displacement attacks are contracts to which one can submit information for a reward [46]. A typical example are bug bounties, where the front-runner can copy the submission from transaction in the transaction pool and submit it itself to steal the reward. Another typical target is registration services that work on the first-come, first-served basis [13]. The front-runner can again simply steal the registration transaction and execute it itself before the original one to reap the benefits.

Displacement attacks are still most known from *arbitrage*, which is type of attack that front-running bots execute practically all the time. Arbitrage is a common activity in traditional finance, so it is no surprise that the same principles also work with cryptocurrencies and *decentralized finance* (DeFi). Quite simply, “arbitrage” means purchasing the same asset in one marketplace and selling it on another simultaneously, with the aim to benefit from the price difference between marketplaces [20]. As DeFi is heavily built onto *decentralized exchanges*, DEXes, which are token exchanges built on smart contracts, it is relatively easy to find and benefit from arbitrage opportunities. What makes DEX arbitrage even more tempting, is that it can be executed *atomically* and totally risk-free. [17] This is done by coding the transactions into a smart contract, and because of how blockchains and smart contracts work, those transactions will always be executed in a *all-or-none* basis [16]. This means that it is impossible for the front-runner to end up with unsold tokens if this kind of proxy contract is used.

To further boost the profits when using proxy contracts, one can use *flash loans*. The basic idea of a flash loan is that borrower does not need to provide any collateral for the loan since it is paid back in the same transaction [50]. For example, with the arbitrage, this can be used in the contract code so that loan is taken right before the arbitrage and paid back right after. Now, since the code is again executed on an all-or-none basis, the lender is guaranteed to get its money back and the borrower gets a massive leverage to the arbitrage with the loan. Flash loans are also commonly used in all sorts of other schemes and attacks to boost the effectiveness.

Compared to displacement attacks, insertion attacks are more similar to what is meant by front-running in traditional finance. Important distinction between displacement attacks and insertion attacks is that in insertion attacks the target transaction have to be executed for front-runner to be able to make profit [13]. When the front-runner’s transaction executes right after the target transaction, the attack is usually referred to as *back-running* instead of front-running [48]. This can still be thought as a sub-case of front-running since the actor front-runs other transactions in the transaction pool and uses information of other transactions for its benefit.

As an example of an insertion attack, consider a scenario where the target transaction wants to buy asset X for some given price. Now, if the front-runner can find the buying opportunity, it can buy the same asset right before the target transaction and sell the asset to the target at a higher price. [13] Other common example is *sandwich trading*. In sandwich trading a searcher tries to find large trades in the transaction pool. If found, it then tries to execute buy order right before the target transaction, and sell right after it to benefit from the rise in the asset price. Insertion attacks in general are riskier than displacement attacks because they cannot be executed atomically. [17]

In addition to previously mentioned examples, there are many other different ways for profiting from transaction ordering dependence. Searchers and miners are also constantly trying to find new MEV opportunities for exploiting other transactions and do front-running to extract maximum profit for themselves.

Mitigations: Better coding practices. To mitigate the exploitation of transaction ordering dependence, a few things can be done. In case of insertion attacks, it is enough to ensure that when the transaction is actually executed, the state of the contract is the same that the sender of the transaction expected it to be. In practice this means no front-runner got to execute in between. This prevention can be done with a little bit of extra information in the contract state and transaction. For example, if the contract state has a counter for total transactions the contract has received, the sending transaction can include the count it expects and the contract can reject the transaction if the two values do not match [14].

Mitigating displacement attacks is a bit trickier and depends on the actual case. When submitting information for a reward, a pattern called commit-reveal scheme is usable [54]. First the submitting party calculates a hash from the submission and some identifying information, and then submits the hash. Only after the hash is mined into a block, the actual submission is revealed. Now anyone can recalculate the hash and check that the party posting the submission really was the original submitter. On the other hand, some things like arbitrage, for example, are practically impossible to counter, even if it would be desirable.

Mitigations: Tools. Front-running and MEV extraction has also some adverse side effects like network congestion and high gas prices because of competing searchers and front-runners. A popular Ethereum project aiming to ease the situation is called *Flashbots* [38]. One of their tools, *MEV-Geth*, expand the Ethereum miner software so that the gas price bidding happens off-chain and in sealed manner. “Sealed” here means that transactions are not exposed in to the public transaction pool or other searchers, which makes generalized front-running impossible [17]. It also eases the congestion since only the successful front-running attempts get mined [38].

3.3.2 Block Stuffing

Block Stuffing is a special subcase of front-running. Instead of gaining benefit from other transactions or being first, the aim is simply to fill blocks with own transactions and prevent other transactions from taking place [13]. This makes block stuffing also a denial of service (DoS) attack.

Block stuffing works because blocks have a predefined maximum size based on network congestion, denominated by *block gas limit* [17]. This is why the attack is also sometimes called *DoS with block gas limit* [46]. The attacker places computationally heavy transactions with high gas price, and essentially pays to fill the block gas limit and to prevent other transactions from taking place. Block stuffing can also happen unintentionally if a smart contract is programmed carelessly. For example, looping over unknown size arrays can end up being really expensive and fill up the block gas limit [46].

A famous attack involving block stuffing was composed against a smart contract game called Fomo3D in 2018 [13]. Fomo3D had a simple idea of rewarding the last address that bought a spot in the game. Price to enter rose every time someone bought a spot and all the buy-ins were added into the grand prize. Each buy-in also added some extra time into a decreasing timer which marked the end of the game. Idea was that after no-one would want to enter anymore and the timer reached zero, the last player who bought a spot would win the game and the prize. The attack itself worked so that after buying the last spot, the attacker filled the following blocks with transactions to specially crafted attacking contracts. This in turn used up all the block gas limit in those blocks preventing anyone else from buying a spot in the game. The attack cost around 10 thousand dollars in the ETH at the time, but allowed the attacker to walk away with around 3 million dollars worth of ETH [51].

As the previous example demonstrates, block stuffing is an effective attack against any contract that requires an action within a defined time limit. Another good example is public auctions, where the highest bid wins after the timer ends. However, block stuffing can be really expensive as the cost of the attack is directly proportional to how long the attacker has to block other transactions, or in other words, it is directly proportional to the number of blocks stuffed [13]. With current gas prices it can be expected to be really expensive, so the reward must also be great for block stuffing to be profitable.

Mitigations: Better coding practices. Since the attack will cost the attacker a lot of money, one good way to avoid getting attacked is to make the attack even more expensive. This can be done, for example, by using bigger time intervals which would require the attacker to stuff more blocks. Another way is to make it harder for other contracts and external parties to access the contract data, so they cannot use it for their benefit. For example, Fomo3D attacker used the game contract to find out whether they were the last participant. Without that information the attack would have been much harder. [51]

To avoid accidental block stuffing via badly designed contracts, one must simply be more cautious. For example, looping over unknown size or dynamic data structures should be avoided [46].

3.3.3 Block Reorganization Attacks

Apart from reordering transactions, another way to create value and do MEV extraction is to reorganize whole blocks. This type of activities can be categorized as *Block Reorganization Attacks*. Successfully reorganizing blocks will naturally require consensus of the network. Obvious way of achieving this is the 51%-attack which was briefly discussed in section 2.2.1. However, instead of actually owning half of the mining power of the network, the attacker can also incentivise other miners to accept the attacker's version of the chain or downright pay the miners to attempt reorganization. Block reorganization attacks will also require forking of the blockchain as the idea is to "rewrite history" by replacing the current tip of the chain by the one the attacker wants. This is why block reorganization attacks also count as *forking attacks*.

First and less serious of the two examples of block reorganization attacks is the *Undercutting Attack*. As with all block reorganization, undercutting attacks can only be performed by miners. If the last mined block is profitable enough, instead of trying to mine the new block after it, a miner can try to "undercut" the last block [8]. It does that by forking the high profit block and trying to mine it again itself. If successful, the miner can then include only some of the high value transactions, leaving some of the profits as incentive for other miners. Other miners can then decide if they want to build onto original chain or the undercutting fork, which will now likely have more profits to take as transaction fees.

In practice, Undercutting Attacks only affect the last block of the blockchain as it is not practical for miners to attempt to mine the fork if the original chain is already ahead. Therefore Undercutting Attacks can also only leverage MEV from new blocks for their benefit. On the other hand, it is also possible to “rewind” history and exploit old MEV opportunities in the old blocks as well as steal the MEV already extracted by others. This is called a *Time-Bandit Attack* [16].

The idea itself is really simple. If the profit from stealing and extracting MEV from the old blocks exceed the rewards for mining a new block and the cost of the attack, the time-bandit attack is feasible [16]. Like with undercutting attack, to be successful, the time-bandit attack requires consensus of the network to accept this new rewound version of the blockchain as a trusted one. Now that many blocks need to be re-mined instead of just the last one, the attacker will need more than just its own computing power and leftover transaction fees to make this happen. In reality, what is needed to alter the consensus is for majority of miners to team up and fork the blockchain from the point wanted and then mine up to the height of the original chain [16].

Example of a successful time-bandit attack can be seen in figure 3.1. In upper diagram a), the miners participating in the attack, colored in red and labeled as “bad”, have forked the blockchain at the block number 9001 and have started to reorganize all the blocks and transactions from that point. Legitimate miners, colored in green and labeled as “legit”, continue to mine the longest version of the blockchain at block number 9003. If the bad miners mine enough blocks so that the fork becomes longer than the original chain, the attack is successful and all the legitimate miners will also switch to the new fork. This can be seen in the lower diagram b). Note that this will render the tip of the original chain and all blocks and transactions in it invalid. In the figure these are blocks number 9002 and 9003. Attackers also reorganize old transactions and place new ones to get maximum profit from attack. For example, second transaction with hash 0x424a is same in both original block 9002 and fork’s block 9002b, but the first transaction before it is changed. Transaction 0x424a can then cause, for example, an increase in a price of an asset and the new transaction 0xb28b executes attacker’s own buy order right before it.

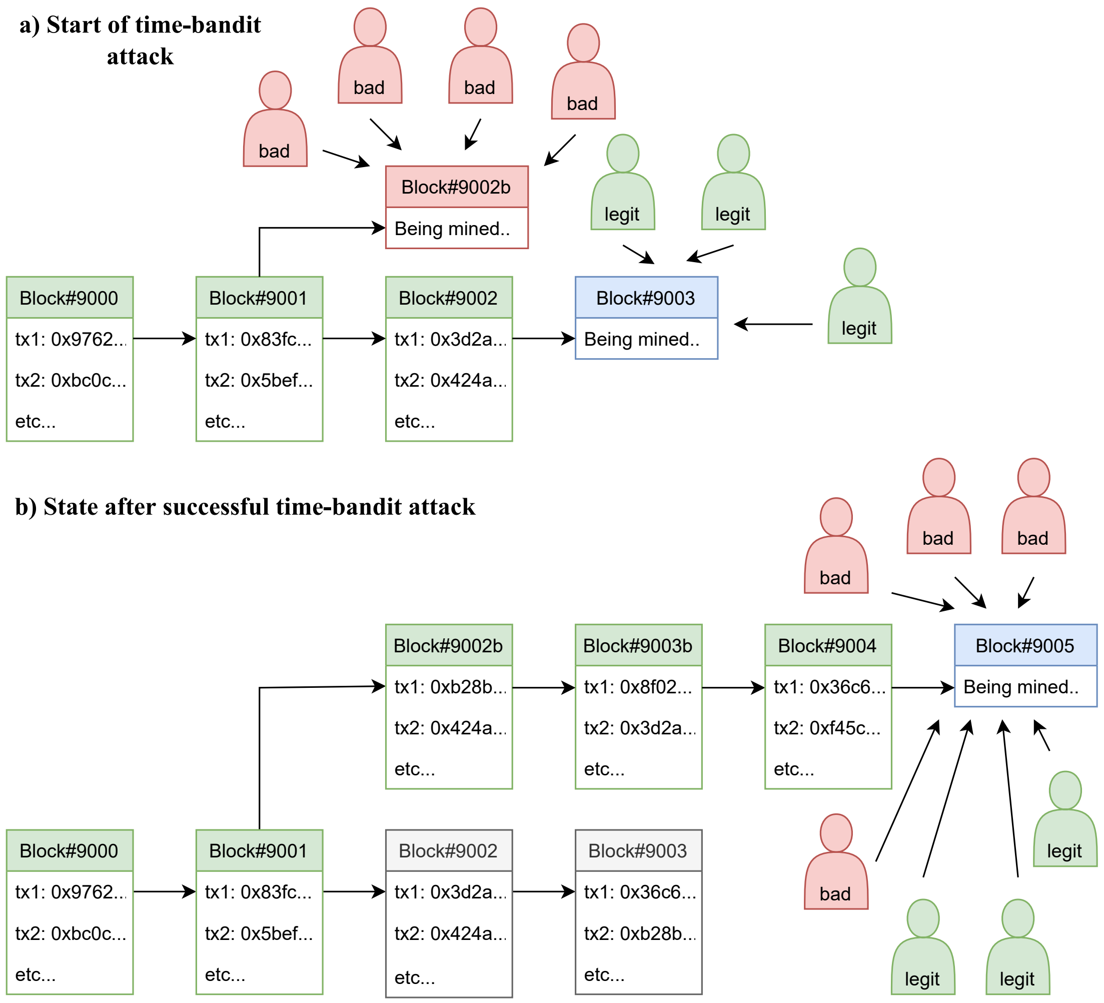

Figure 3.1: Phases of successful time-bandit attack. Diagram a) shows the starting state where miners participating in the attack fork the blockchain at the wanted point and start to mine this new fork. Diagram b) shows the state after successful attack, where the attacker's fork has become the longest one. Now all miners accept it as the most trusted version of the blockchain and start mining new blocks to it.

Previously, the risk of time-bandit attack happening was considered slim, since it was thought that miners would not want to destabilize the whole protocol and tank the price of the token by participating in the attack [28]. After all, it is the miners who benefit the most from the healthy protocol via block rewards and fees. However, minor time-bandit attacks have already happened and nowadays even tools have been built that allow paying miners for an attack via smart contracts in case the attack were to be successful [26]. One example of such a tool, *Request for Reorg* [1], works by locking in some amount of ETH as reward in a smart contract with the rewind request itself. Then, when miners rewind the blockchain, they include this request transaction as the first one (because the blockchain state is rolled back), and when they have mined enough blocks on this new fork, they can claim the reward. For the rewards to be of any use, the rewound fork must also become the longest and therefore most trusted version of the blockchain. This adds an incentive for participating miners to keep mining blocks to this new fork. It is also good to note that it is not possible to rewind past the block containing the reward contract deployment as the reward function itself would not exist in that case.

To give an idea of what a real life time-bandit attack could look like, the authors of the famous "Flash Boys 2.0" -article [16] gave a theoretical example of MEV opportunity via DEX rewinding: Considering a scenario where the price of an token rose from 1 USD to 3 USD with 1M USD in trade volume, a theoretical 2M USD in MEV can be made with the price difference (1M * (3 USD - 1 USD) = 2M USD). To do so, the attacker can buy mining power to rewind the history so that it is on the buy-side of the trades before the spike, netting the 2M USD gross profit for itself. If the cost of buying the mining power is less than 2M USD, the attacker makes profit.

Nowadays the scale of time-bandit attack would most likely be much larger since the cost of buying the required mining power is really high. Tracking site for 51%-attack costs, Crypto51, estimates that renting enough mining power for an one hour attack on Ethereum with prices at the end of January 2022 would cost around 1,5M USD [15]. The attack would also require that enough miners were willing to participate in this kind of action. From this it follows that time-bandit attacks are more likely to happen on smaller protocols since there are less miners and it is cheaper to buy the mining power. This has been seen in practice since there have recently been successful attacks on smaller chains like Bitcoin Gold [15]. A big portion of mining power rental cost can, however, be subsidized by stealing all the other MEV opportunities and block rewards from the past blocks, so attacks on big protocols like Ethereum are still possible [16]. Also the built-in decrease in block mining rewards makes it increasingly compelling for miners to participate in this kind of MEV extraction [19][8].

Mitigations: Technology changes. Out of all attacks explained in this thesis, block reorganization attacks pose the biggest threat to any blockchain protocol as they directly compromise the whole idea of blockchain immutability. Time-bandit attack is also such a large-scale attack that the mitigations must be done on the technology level.

The most prominent mitigation is moving away from Proof of Work -algorithms, where miners have the power to do block reorganization. For example, at the time of writing this, Ethereum is planning on migrating to Proof of Stake -type of consensus algorithm. In Ethereum's version of Proof of Stake, a pseudorandomly sampled set of nodes is picked as a committee, where one node proposes a new block and rest of the committee validates it [17]. As the committee is picked randomly from all the nodes, it is really hard for an attacker to get majority of the validators and perform even one block reorganization [27]. This makes undercutting attacks much harder than in Proof of Work based protocols. Time-bandit attacks are also much harder since blocks deeper than 64 blocks in the blockchain are considered "finalized" and truly immutable, so any conflicting blocks would cause a broken system [27]. Reorganizing up to past 64 blocks is still possible, but it would require that enough nodes controlled by the attacker are randomly picked so that they form the majority of validators for all those blocks. To further discourage bad behaviour, Ethereum's Proof of Stake uses a *slashing* mechanism that penalises those validators who do not follow the rules. If validator is caught on behaving badly, it can lose part of or all of its *stake* which is the amount of ETH they locked in as collateral in the protocol when becoming a validator [6].

Another way, closely related to the Proof of Stake, is called *MEV smoothing*. Loosely speaking, this means reducing the variance of extracted MEV so that validator rewards are as uniform as possible [21]. This would in turn discourage validators for misbehaving for their own benefit when they compete for the best rewards.

There are also a few extra things that can be done to mitigate undercutting attacks. First one, already implemented in Bitcoin, is adding a parameter for required block height to a transaction [37]. This makes it harder to include high fee transactions in an undercutting fork as the height of the fork would need to match the block height set in the transaction. Another miner specific mitigation measure is to set aside some free-to-claim value in the block for other miners to grab to discourage undercutting attempts [37]. In a sense, this is equivalent for the miner undercutting itself.

3.4 Integer Overflow and Underflow

Integer Overflow and *Integer Underflow* are fairly simple vulnerabilities and very common in conventional coding. With smart contracts the idea is the same. As variables in any system have a maximum number of bits they can store, all variables will also have a maximum and minimum integer value they can represent. When a value of variable goes over the maximum or under the minimum via arithmetic operations, if the code is vulnerable, the value will "loop back". This means overflow will loop back to minimum value and underflow to the maximum value. [40]

In Ethereum, integers can be up to 256 bits long. With Solidity, it is also possible to define integers in smaller sizes starting from 8 bits [45]. Integers can be signed or unsigned, which simply defines if negative integers are allowed or not. Signing does not affect overflow mechanics per se, but it changes the maximum and minimum values that can be expressed and therefore also the values where integer loops around.

Overflows and underflows have been a real issue for smart contracts. For example, in 2018, an Ethereum smart contract and token called BeautyChain, (BEC), got attacked with overflow attack where the attacker stole a high amount of tokens via an overflow bug in the contract [22]. The vulnerable function itself can be seen in the listing 3.3.

```
1 function batchTransfer (address[] _receivers, uint256 _value) public
    whenNotPaused returns (bool) {
2     uint cnt = _receivers.length;
3     uint256 amount = uint256(cnt) * _value;
4     require(cnt > 0 && cnt <=20);
5     require(_value>0 && balances[msg.sender]>=amount);
6     ...
7     return true;
8 }
```

Listing 3.3: Vulnerable function in BeautyChain (BEC) smart contract. [22]

The vulnerability itself lies in the fact that anyone can fully control the calculated variable `amount` on line 3 by simply setting the function parameters `_receivers` and `_value`. This is because it is simply the multiple of `_value` and number of addresses in the `_receivers` -array. In the case of the actual attack, the attacker chose two addresses it controlled and the value 0x8000...000 (63 0s). Now, when the `amount` was calculated, it caused an overflow because it is actually one over the maximum value that 256-bit integer can hold. Because the type is unsigned integer, the `amount` incorrectly looped back to zero and the balance check on line 5 passed. This caused the `_value` amount of tokens to be sent to both attacker-controlled addresses even when there was no balance.

Mitigations: Tools. A good way to battle overflows and underflows is to use libraries that check for too big or small integers automatically [13]. This way the invalid value will cause a run-time error if it does not fit the variable and exploitation will be impossible. This is equivalent to Java integer overflow errors, for example. There is also tools that check for overflow and underflow vulnerabilities on existing smart contracts [22][43]. However, at least for Ethereum smart contracts and when coding in Solidity, these kind of tools are nowadays mostly redundant. This is because, as of version 0.8.0, Solidity automatically checks arithmetic operations for overflows and underflows and returns run-time assertion error if found [45]. This will also revert the running transaction. Automatic checking can still be disabled if so wanted.

Mitigations: Better coding practices. One can also take caution when coding, to prevent overflows and underflows. For example, one good practice is paying attention to who has an authority to make changes to the given variable. If anyone can make the changes, there is a higher risk of exploitation. It is also good to think of how the value actually changes and if there is a risk of reaching maximum or minimum value. In the case of maximum value, smaller data-types like `uint8` or `uint16` are also more prone to exploitation since the maximum value is more easily reached.

3.5 DoS With Revert

One specific characteristic of smart contract capable blockchains is that if a transaction fails for one reason or another, the protocol state must be reverted to the stable one right before the transaction. This is what is meant by *atomic* transactions, meaning they either execute fully or not at all. Reason for this is that the blockchain state must always be handled in a deterministic manner for the consensus algorithm to work and therefore the state cannot be ambiguous.

An attacker can leverage this reverting property by deliberately causing a transaction to fail and preventing a smart contract from functioning properly. This may also prevent other users from using the smart contract, which is why the attack is called *Denial of Service With Revert* or *DoS With Revert*. As with block stuffing (section 3.3.2), the contract malfunction can of course happen also by accident if a transaction fails unintentionally [42].

To cause a transaction to fail, the attacker must get control of the execution. The best way of doing this is by utilizing the fallback function of smart contracts [42]. This is done in the same manner as explained when discussing the re-entrancy vulnerability in section 3.1. This is also the most common way of getting control since revert vulnerability is usually found in contracts which send ETH to user defined addresses, and fallback function is always triggered if a contract receives ETH [41]. Another way to get control would be if the target contract called random contracts and their functions, but this does not seem plausible.

To cause the revert, the attacker can craft a malicious smart contract which has a fallback function that will always throw an error. Then when the target contract wants to send ETH to the malicious contract's address, the fallback function will be called and the transaction will always fail and revert because of the error. If the ETH transfer must be successful for contract to function properly, this will also cause a denial of service [42].

To give an example, consider a batch refund function seen in listing 3.4. The `refundAll` function iterates through a list of addresses which will be refunded their full balance based on some internal state. However, as function execution and transactions are atomic and the `require` function will throw an error on failure, even if only one of the `send` calls on line 3 fails, the whole execution will fail and none of the addresses will receive any funds. All the attacker has to do is to get the address of the malicious smart contract in the `_refundAddresses` -array to cause everyone else to be incapable of receiving refunds. The malicious attack contract itself can simply use `throw` in fallback function to manually throw an error and revert the whole execution.

```
1 function refundAll(address[] _refundAddresses) public {
2     for(uint i; i < _refundAddresses.length; i++) {
3         require(_refundAddresses[x].send(balances[_refundAddresses[x]]))
4     }
5 }
```

Listing 3.4: Refund function with revert vulnerability.

Mitigations: Better coding practices. Since it is always possible that a transaction fails, even by accident, it is not recommended to build contracts that depend on successful transactions to uncontrolled addresses. Therefore, the same principle as with re-entrancy applies here, that if avoidable, smart contracts should "neither send ETH nor call untrusted contracts" [17]. One good way to avoid calling untrusted contracts or rely on sending ETH is by isolating external calls to own transactions which users can then initiate. Then they can safely fail without affecting anything else. [46] This practice is often referred as *pull over push*, since it is usually done with payments where users "pull" the funds by withdrawing instead of contract "pushing" them via sending.

If external calls or sending funds must still be done, there are some good practices to mitigate the risk. First, one should avoid combining multiple calls into a single transaction, especially when they are executed as part of a loop [46]. This was the mistake done in listing 3.4, which caused one failing call to fail all the others. This mistake also proves the second rule that one should always assume that external calls can fail. Therefore it is vital to implement logic to handle failed calls [46]. This is equivalent to error handling in conventional coding.

3.6 Insufficient Gas Griefing

Insufficient Gas Griefing is an attack on contracts that accept data and use this data in a sub-call on another contract [46]. The target of the attack is the party that submits the data and the attacker is the one placing the transaction which would execute the sub-call. The basic idea is that if the contract is vulnerable and not enough gas is provided, the sub-call will fail but the attackers transaction will not. This can in turn result in an unwanted state where data gets passed but the intended underlying call will fail.

A typical example of this type of vulnerable contract is a *relayer contract* which allows executing transactions on behalf of another party [13]. This can be desired, for example, if the original user does not have enough gas to execute the transaction by itself. The user can then pass the wanted transaction off-chain to some other user, which will in turn call the relay contract with the transaction data that original user has signed. The relay contract will record the data and try to execute the relayed transaction as a sub-call. If the sub-call fails and failure is not handled, i.e., the relay contract execution continues and does not revert, the relay contract can end up in a state where relayed transaction is marked as executed by the relay contract when it has in fact failed. The attacker can then utilize this misbehaviour by providing just enough gas to pass on the signed data but fail the sub-call, thus essentially preventing transactions from happening. [46]

The attack does not directly benefit the attacker in any way and it will cost gas [23]. However, it might still have some indirect benefits by preventing other users from placing transactions. Even if not beneficial, it still causes harm, or *grief*, to the victim as it makes it harder for them to use the system. This is why the attack is called insufficient gas griefing, or simply *Griefing* [13]. Preventing other users from using the system as intended makes insufficient gas griefing also a denial of service -attack.

Mitigations: Better coding practices. First thing that can be done to prevent griefing is to make sure that the party that forwards the transaction has enough gas [13]. This can be done by checking the gas left against an user set gas limit in the sub-call function. If the underlying contract cannot be changed, then one solution is to only allow execution for trusted users [46]. This check can be done in the relay contract.

3.7 Forcibly sending Ether To A Contract

A common misconception with Ethereum smart contracts is that they can only obtain ETH via functions defined as *payable* [33]. Payable functions also include fallback function which is typically executed when a contract receives ETH and was already discussed with re-entrancy (section 3.1) and DoS with revert (section 3.5). However, there are ways to force ETH balance on to contracts without executing any code on them.

The first way force ETH to a target contract is to craft a contract with a built-in *selfdestruct* function [33]. When a contract uses selfdestruct with a defined target address, the contract code will be wiped from the contract address and its ETH balance will be sent to the target address. If the target address belongs also to a smart contract, this ETH transfer will not trigger that contracts fallback function. Now, to force ETH onto target contract, one can simply make own contract, send ETH to it, and then call selfdestruct with the target contract's address.

Another way to force ETH on to the target contract is to pre-load its address with ETH before the contract is even deployed [33]. Addresses are calculated in a deterministic way so it is possible to calculate the contract address beforehand, send ETH to it, and cause the contract to have a non-zero balance when created.

Where this forcibly receiving ETH might become problematic, is when a smart contract assumes a specific ETH balance and that it can only be altered via contract functions [46]. For example, consider a hypothetical case where a contract assumes that its balance increases and decreases in fixed size increments defined by contract functions. Let us assume it can only change in 1 ETH steps. Now, if the exact values are used in conditional checks in the contract logic, the malicious party can force some fraction of ETH on the contract balance, thus making reaching the exact value via contract functions impossible. In the worst case, this type of situations can lead to total denial of service where the whole contract is rendered unusable because of an unexpected balance [46].

Mitigations: Better coding practices. The mitigation for this is simple. One should simply avoid using strict equality checks against contract balance [46]. If condition checks must be used, the unexpected balances should be accounted for [33].

3.8 Oracle Manipulation

The last discussed vulnerability is possible because of the deterministic nature of the blockchain. Smart contracts cannot inherently interact with the outside world because it would make execution ambiguous and unpredictable. Therefore some interface is needed if any off-chain data is wished to be used. *Oracles* are designed to solve this problem and act as an interface service between data source and smart contracts [9]. Serving data from outside world to smart contracts is the main use case, but oracles can also provide information from the blockchain itself, such as token prices in the *decentralized exchanges* (DEXes).

All oracles are smart contracts which provide some useful data that is kept up to date. They can be *off-chain* or *on-chain*. Off-chain means that the data that oracle serves comes from outside of the blockchain, and respectively on-chain means the data comes from inside the blockchain. Oracles can also be centralized or decentralized, which simply means difference between central authority that keeps data up to date or if anyone can participate in updating the data. [18]

If a malicious party can manipulate an oracle and a smart contract relies on its data, the consequences can be bad [13]. A typical example smart contracts that are oracle dependent are DeFi contracts that rely on token price data. Being able to manipulate price information of the tokens that the project uses can lead to exploitation and massive financial losses. [18]

The method of the manipulation depends on the type of the oracle in question. Off-chain oracles that use off-chain data feeds can be manipulated by attacking the data source. Centralized oracles, on the other hand, can be manipulated by hacking the central authority. This can be via leaked password or a hacked wallet, for example. From the blockchain perspective the interesting type of oracles are however the decentralized on-chain oracles, since they can be more or less manipulated by anyone and everything happens on-chain.

Typical manipulation attack is done against a DeFi project that uses price data calculated from a DEX. In this type of attack a Defi project uses a DEX as a decentralized on-chain oracle [18]. DEXes typically allow trading of tokens via *token-pair liquidity pools*, so it is easy to calculate prices for tokens. Token-pair liquidity pool simply means a pool of cryptocurrency tokens which a smart contract holds. That smart contract then allows users to trade these tokens against a calculated price based on the ratio of the tokens in the pool.

For example, if pool is with ETH/USD -pair and there are 1 000 ETH and 20 000 USD in the pool, the price of ETH can be calculated as (20 000/1 000) USD = 2 000 USD. However, as anyone can affect the amount of tokens in the liquidity pool by simply trading them, anyone can also easily manipulate the price in this case [18].

There are many ways to benefit from the manipulation, and in practice the effect depends on the target smart contract. One example is lending protocols which require placing tokens as collateral to cover the losses if the borrowed money cannot be paid back or prices change too much [18]. If the attacker can manipulate the token price when the loan is taken, it can make the collateral seem much more valuable than its true value. This allows the attacker to borrow much more than the initial value of the collateral, and in practice steal money. Other common manipulation exploits include arbitrage, liquidations, DoS, etc. [13].

Mitigations: Better coding practices. First mitigation that can be applied when using on-chain decentralized oracles is some sort of validation on the data. For example, calculated price data can be compared against previously known good rates [18]. If DEXes are used as a data source, as was the case in the example before, it is also good to make sure that there is enough liquidity in the liquidity pool [18]. This makes price manipulation via trading harder and more expensive as the manipulator would have to do bigger trades to have meaningful effect on the price data. One can also make an oracle a bit more centralized and trustworthy by allowing only a trusted set of accounts to update it [18]. This would mean that the attacker would have to wait for the authorized account to update the manipulated data before they can execute the exploit. There is however an disadvantage in that this will likely decrease update frequency.

Mitigations: Tools. Existing tools and oracles can be used to mitigate the manipulation risk. Many of them rely on time-weighted averages and multiple sources to make it almost impossible to manipulate the data [13]. One such an oracle is called *Uniswap Time-Weighted Average Price* (TWAP), provided by DEX market leader Uniswap [18]. Many of these oracles also rely on the trusted set of users to update the data [18]. One popular oracle like that is *Chainlink*, which has a set of nodes that update token prices and after the update the oracle aggregates the data. In this approach one must however trust third parties to be honest with the updated prices. In the end many of these solutions have their own pros and cons and there is always a trade-off between update frequency and correctness of the data. Open and decentralized solutions where everyone can update the data are much faster to update, but also easier to manipulate since changes propagate right away. Finding to right tool for the job then boils down to the actual use case.

4 Discussion

Now that all the covered attacks and vulnerabilities are explained, it is useful to categorize and analyze them a bit to get a better understanding of the bigger picture of smart contract attacks and their related security issues.

First way to create a categorization is by the place where the actual vulnerability or property lies that allows exploitation. This classification was already presented at the start of chapter 3, but to recap, the vulnerability itself may be in smart contract code, it may be in some property of the blockchain technology, or the exploitation may need both combined. Attacks leveraging both the technology properties and vulnerable smart contract code are *Re-entrancy*, *Timestamp Dependence*, *Transaction Ordering Dependence*, *DoS With Revert*, *Insufficient Gas Griefing*, *Forcibly sending Ether To A Contract* and *Oracle Manipulation*. Attacks leveraging only the underlying blockchain technology and its properties are *Block Stuffing* and *Block Reorganization Attacks*. And the only attack that exploits just the contract code is *Integer Overflow and Underflow*. This categorization can be seen in the middle column of the table 4.1.

Majority of the discussed attacks leverage both technology properties and vulnerable smart contract code. This implies that blockchain and smart contract technologies bring up totally new kind of challenges that make designing secure smart contracts hard. Many of the vulnerabilities from conventional coding are also present, which is understandable because smart contract capable technologies are usually Turing-complete. These more conventional vulnerabilities only lie in smart contract code and are not affected by blockchain technology per se. Integer overflow and underflow are examples of this case.

On top of security issues in the code or in the technology, the immutability of the blockchains makes patching badly designed code really hard. Therefore it is vital to pay extra attention to security when designing smart contracts. [24]

One useful way to look at software security is via its lifecycle. With smart contracts it can be roughly split into *security design*, *security implementation*, *testing before deployment* and *monitoring and analysis* [24].

Security design is done before anything is implemented. It includes all kinds of useful coding practices and principles that are can be good against specific problems or in designing

Attack/Vulnerability	Vulnerability location	Mitigations*
Re-entrancy	Code/Technology	BCP/T/TC
Timestamp dependence	Code/Technology	BCP/TC
Transaction Ordering Dependence	Code/Technology	BCP/T
Block Stuffing	Technology	BCP
Block Reorganization Attacks	Technology	TC
Integer Overflow and Underflow	Code	BCP/T
DoS With Revert	Code/Technology	BCP
Insufficient Gas Griefing	Code/Technology	BCP
Forcibly Sending Ether to a Contract	Code/Technology	BCP
Oracle Manipulation	Code/Technology	BCP/T

Table 4.1: Attacks and vulnerabilities categorized by location of the vulnerability and by possible mitigations. *Mitigation abbreviations: BCP = Better Coding Practices, T = Tools, TC = Technology Changes.

smart contracts in general. As an example, consider *pull over push* -pattern against Dos With Revert and Re-entrancy or *preparing for failure* when coding smart contracts in general. Many of the patterns and principles suggested against attacks in the chapter 3 belong to the security design phase. [24]

Security implementation happens in parallel to the smart contract implementation as it is tied within the actual coding process. Apart from following practices and implementing design patterns, automated tools can be used at this point to help in creating secure code. These tools include things like analysis tools, IDE expansions and secure libraries, for example. Both implementation phase and design phase can also include security modeling, which aims to prove the correctness of the code and help in eliminating errors. [24]

Tools are often used also in the last two lifecycle phases, testing before deployment and monitoring and analysis [24]. Therefore it is safe to say that proper smart contract security uses all the same practices that are familiar from conventional coding. However, the security is usually much more in focus with smart contracts as they are immutable, visible for all, and often deal with huge amounts of capital as tokens [49].

Apart from coding practices and tools, smart contract security can also be improved by making changes to the underlying blockchain technology. In the best case, changes to the underlying technology can eliminate a security issue altogether. For example, timestamp dependence exploits can be fully mitigated by hiding block timestamp from smart contracts altogether so it cannot be used. Technology changes can also be partially effective in mitigating a security issue. One example of this is mitigating block reorganization attacks by moving to the Proof of Stake -algorithm, which was explained in the section 3.3.3. However, in practice changes to whole blockchain protocols are really hard to implement if the blockchain has already been launched, because the majority of the nodes would need to switch to a new version before the change can happen. Also changes cannot be too restrictive as it would limit the usefulness of smart contracts. In some special cases, tools can be used to mitigate the problem without the need for bigger changes. Flashbots expansion for Ethereum nodes is one such tool, which aims to help with front-running problems. It was briefly explained in the section 3.3.1.

Based on the smart contract security concerns above, three classes of threat mitigation are identified: *better coding practices*, *automated tools* and *technology changes*. Therefore attacks in chapter 3 can also be indirectly categorized with the suggested mitigations against them. Note that this categorization is not complete by any means and it is likely that mitigations from each of these classes can to some extent be used against all threats. However, this categorization still represents suggested types of mitigations against given threat at the time of writing. These categories can be seen in the third column of the table 4.1. For the abbreviations in the mitigation column, “BCP” means *better coding practices*, “T” means *tools* and “TC” means *technology changes*.

One point that stands out is that all the discussed attacks except block reorganization attacks can be mitigated by following better coding practices, at least to some extent. This seems to indicate that it is crucial for developers to pay attention to security issues in the design and implementation phases. This is also in line with earlier research where it was found out that smart contract developers tend to value security more than practitioners in other software areas [49].

Tools and technology changes can be useful against specific threats. Tools will most likely be even more widely used when more sophisticated and advanced tools are designed. Development of smart contracts can then move more towards conventional coding where much of the security issues are covered via automated tools like scanners and analyzers. On the other hand, technology changes can often be something that is not wanted even if those changes would counter some threats. This is because the changes might restrict smart contracts too much, they might be too hard or laborious to implement, or there might simply be a better and easier solution to the problem they would mitigate. The most notable technology change that will counter some issues is moving away from the Proof of Work -consensus algorithm. However, even this change is mainly done because of energy usage and scaling issues, not because of security issues. Therefore, it can be said that most smart contract security issues are not something that underlying blockchain technology should address and those issues should be addressed in the smart contract code instead.

4.1 Future Views

As one of the hottest and most prominent topics in the tech industry, blockchain technologies seem to have a very bright future ahead. Therefore, the surge in smart contracts and their adaptation is also expected to continue. This naturally brings up many new topics to be studied and discussed, along with new security issues.

The most urgent topic at the time of writing is probably the Proof of Stake -algorithm, as Ethereum is currently in the middle of the migration process to change their consensus algorithm to it. At the same time Ethereum community is also designing and implementing so called “*sharding*”, which simply means splitting blockchain to multiple “shards” to make it scale and perform better [17]. As these changes make the protocol more complicated, they will no doubt bring up new security challenges. This is an area that will need further study.

Another hot topic with smart contracts is MEV extraction. One cornerstone idea behind blockchain technologies is game theory, which in the blockchain context means that the “rules” of the protocol should be made so that behaving correctly is always the optimal choice for everyone. In other words, it is assumed that users behave selfishly so selfish acts should be allowed and the protocol designed so that selfish act is also the optimal act. This also means that MEV extraction, as a selfish act, is expected, and should be mitigated on the technology level instead of being regulated or punished outside the protocol. This will bring up a lot of new topics when new types of MEV extraction and their mitigations are invented.

Outside of strictly technical topics, regulation and legal issues of cryptocurrencies are also discussed actively. El Salvador made Bitcoin a legal tender as the first country in the world in June 2021 [47]. At the same time many countries are discussing how cryptocurrencies should be regulated and some countries have already banned them, most notably China [5]. As huge amounts of money are invested and involved in the cryptocurrencies, the need for better understanding is urgent. Therefore the whole cryptocurrency space has massive research potential also in other fields like economy, politics or sociology.

5 Conclusions

Smart contracts expand the utility of blockchains by allowing storing and executing code in the blockchain. This brings up many new interesting use cases when on top of value transactions, users can also verify and execute deployed code. This added utility also brings up many new security challenges which are of the utmost importance with smart contracts since a lot of money is involved.

In this thesis we give an up-to-date review on smart contract security issues. We discuss the most common attacks and vulnerabilities as well as suggested mitigations against them. Then the findings are summed up and analysed to give a big picture of the current state of smart contract security. We find out that many of the attacks could be avoided or at least severely mitigated if the developers followed good coding practices and used design patterns that are proven to be good. Another finding is that changing the underlying blockchain technology to counter the issues is usually not the best way, as it is hard and troublesome to do and might restrict the usability of contracts too much. Issues may also be so specific to the use case that changing underlying technology would be an overkill. However, many of the attacks still leverage the behaviour of the underlying technology in addition to the vulnerable smart contract code, so the technology aspect should not be overlooked. Lastly, we find out that new automated tools for security are being developed and emerging all the time, which indicates movement towards more conventional coding where automated tools like scanners and analysers are being used to cover a large set of security issues.

As a new topic under active development, smart contract security will no doubt need more research in the future. This is especially true since smart contracts find new use cases all the time and security issues will no doubt emerge along them. Also, the way the protocols and smart contracts are designed in the online communities brings up its own challenges since, as discussed in the thesis, the scientific field is now the one that has to keep up with the real world and not the other way around. Therefore it is vital for smart contract security research to also keep up with the pace.

Bibliography

[1] 0xbunnygirl. *Request for Reorg (RFR)*. GitHub repository. [accessed 26-May-2022]. URL: https://github.com/0xbunnygirl/request-for-reorg.

[2] M. Andoni, V. Robu, D. Flynn, S. Abram, D. Geach, D. Jenkins, P. McCallum, and A. Peacock. "Blockchain technology in the energy sector: A systematic review of challenges and opportunities". In: *Renewable and Sustainable Energy Reviews* 100 (2019), pp. 143–174. DOI: 10.1016/j.rser.2018.10.014.

[3] N. Atzei, M. Bartoletti, and T. Cimoli. *A survey of attacks on Ethereum smart contracts (SoK)*. Vol. 10204 LNCS. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2017, pp. 164–186. DOI: 10.1007/978-3-662-54455-6_8.

[4] L. M. Bach, B. Mihaljevic, and M. Zagar. "Comparative analysis of blockchain consensus algorithms". In: *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2018 - Proceedings*. 2018, pp. 1545–1550. DOI: 10.23919/MIPRO.2018.8400278.

[5] P. Bajpai. *Countries Where Bitcoin Is Legal and Illegal*. News article. [accessed 20-March-2022]. 2021. URL: https://www.investopedia.com/articles/forex/041515/countries-where-bitcoin-legal-illegal.asp.

[6] V. Buterin, D. Reijsbergen, S. Leonardos, and G. Piliouras. "Incentives in ethereum's hybrid casper protocol". In: *ICBC 2019 - IEEE International Conference on Blockchain and Cryptocurrency*. 2019, pp. 236–244. DOI: 10.1109/BLOC.2019.8751241.

[7] V. Buterin. *Ethereum Whitepaper*. Whitepaper. ethereum.org, 2013. URL: https://ethereum.org/en/whitepaper/.

[8] M. Carlsten, H. Kalodner, A. Narayanan, and S. M. Weinberg. "On the instability of Bitcoin without the block reward". In: *Proceedings of the ACM Conference on Computer and Communications Security*. Vol. 24-28-October-2016. 2016, pp. 154–167. DOI: 10.1145/2976749.2978408.

[9] Chainlink. *What Is a Blockchain Oracle?* Web page. [accessed 26-May-2022]. URL: https://chain.link/education/blockchain-oracles.

[10] Y. Chen and C. Bellavitis. “Blockchain Disruption and Decentralized Finance: The Rise of Decentralized Business Models”. In: *Journal of Business Venturing Insights* 13 (2020), e00230. DOI: 10.1016/j.jbvi.2019.e00151.

[11] K. Christidis and M. Devetsikiotis. “Blockchains and Smart Contracts for the Internet of Things”. In: *IEEE Access* 4 (2016), pp. 2292–2303. DOI: 10.1109/ACCESS.2016.2566339.

[12] CoinGecko. *Top Smart Contract Platform Coins by Market Capitalization*. Web page. [accessed 26-May-2022]. URL: https://www.coingecko.com/en/categories/smart-contract-platform.

[13] ConsenSys. *Ethereum Smart Contract Best Practices*. Web page. [accessed 26-May-2022]. URL: https://consensys.github.io/smart-contract-best-practices/attacks/.

[14] C. Coverdale. *Solidity: Transaction-Ordering Attacks*. Blog post. [accessed 26-May-2022]. 2018. URL: https://medium.com/coinmonks/solidity-transaction-ordering-attacks-1193a014884e.

[15] Crypto51. *Ethereum (ETH)| Crypto51*. Web page. [accessed 26-May-2022]. URL: https://www.crypto51.app/coins/ETH.html.

[16] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels. “Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability”. In: *Proceedings - IEEE Symposium on Security and Privacy*. Vol. 2020-May. 2020, pp. 1106–1120. DOI: 10.1109/SP40000.2020.00040.

[17] ethereum.org. *Ethereum development documentation*. Protocol documentation. [accessed 26-May-2022]. URL: https://ethereum.org/en/developers/docs/.

[18] Extropy.IO. *Price Oracle Manipulation*. Blog post. [accessed 17-February-2022]. 2021. URL: https://extropy-io.medium.com/price-oracle-manipulation-d46fd413cc17.

[19] A. Feign. *How Much Energy Does Bitcoin Use?* News article. [accessed 23-May-2022]. 2021. URL: https://www.coindesk.com/business/2021/08/18/how-much-energy-does-bitcoin-use/.

[20] J. Fernando. *Arbitrage*. Investopedia article. [accessed 26-May-2022]. 2022. URL: https://www.investopedia.com/terms/a/arbitrage.asp.

[21] fradamt. *Committee-driven MEV smoothing*. Blog post. [accessed 02-February-2022]. 2021. URL: https://ethresear.ch/t/committee-driven-mev-smoothing/10408.

[22] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen. "EASYFLOW: Keep ethereum away from overflow". In: *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion, ICSE-Companion 2019*. 2019, pp. 23–26. DOI: 10.1109/ICSE-Companion.2019.00029.

[23] R. Horrocks. *What does griefing mean?* Forum post. [accessed 23-May-2022]. 2018. URL: https://ethereum.stackexchange.com/questions/62829/what-does-griefing-mean.

[24] Y. Huang, Y. Bian, R. Li, J. L. Zhao, and P. Shi. "Smart contract security: A software lifecycle perspective". In: *IEEE Access* 7 (2019), pp. 150184–150202. DOI: 10.1109/ACCESS.2019.2946988.

[25] IBM. *What is blockchain technology?* Web page. [accessed 23-May-2022]. URL: https://www.ibm.com/topics/what-is-blockchain.

[26] C. Kim. *Valid Points: The Problem With MEV on Ethereum*. News article. [accessed 26-May-2022]. 2021. URL: https://www.coindesk.com/tech/2021/07/14/valid-points-the-problem-with-mev-on-ethereum/.

[27] G. Konstantopoulos and V. Buterin. *Ethereum Reorgs After The Merge*. Blog post. [accessed 01-February-2022]. 2021. URL: https://www.paradigm.xyz/2021/07/ethereum-reorgs-after-the-merge#post-merge-ethereum-with-proof-of-stake.

[28] G. Konstantopoulos and L. Zhang. *Ethereum Blockspace - Who Gets What and Why*. Blog post. [accessed 26-May-2022]. 2021. URL: https://research.paradigm.xyz/ethereum-blockspace.

[29] A. Kosba, A. Miller, E. Shi, Z. Wen, and C. Papamanthou. "Hawk: The Blockchain Model of Cryptography and Privacy-Preserving Smart Contracts". In: *Proceedings - 2016 IEEE Symposium on Security and Privacy, SP 2016*. 2016, pp. 839–858. DOI: 10.1109/SP.2016.55.

[30] P. Kostamis, A. Sendros, and P. Efraimidis. "Exploring Ethereum's Data Stores: A Cost and Performance Comparison". In: *2021 3rd Conference on Blockchain Research and Applications for Innovative Networks and Services, BRAINS 2021*. 2021, pp. 53–60. DOI: 10.1109/BRAINS52497.2021.9569804.

[31] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. "Making smart contracts smarter". In: *Proceedings of the ACM Conference on Computer and Communications Security*. Vol. 24-28-October-2016. 2016, pp. 254–269. DOI: 10.1145/2976749.2978309.

[32] S. MacKenzie. *Bitcoin's biggest upgrade in four years just happened – here's what changes*. News article. [accessed 23-May-2022]. 2021. URL: https://www.cnbc.com/2021/11/14/bitcoin-taproot-upgrade-what-it-means-for-investors.html.

[33] A. Manning. *Solidity Security: Comprehensive list of known attack vectors and common anti-patterns*. Blog post. [accessed 16-February-2022]. 2018. URL: https://blog.sigmaprime.io/solidity-security.html#ether.

[34] M. I. Mehar, C. L. Shier, A. Giambattista, E. Gong, G. Fletcher, R. Sanayhie, H. M. Kim, and M. Laskowski. "Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack". In: *Journal of Cases on Information Technology* 21 (1 2019), pp. 19–32. DOI: 10.4018/JCIT.2019010102.

[35] C. Mitchell. *Front-Running*. Investopedia article. [accessed 26-May-2022]. 2022. URL: https://www.investopedia.com/terms/f/frontrunning.asp.

[36] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Whitepaper. bitcoin.org, 2009. URL: https://bitcoin.org/bitcoin.pdf.

[37] F. Nieto. *Can Undercutting attacks be Mitigated?* Forum post. [accessed 23-May-2022]. 2019. URL: https://bitcoin.stackexchange.com/questions/67697/can-undercutting-attacks-be-mitigated.

[38] A. Obadia. *Flashbots: Frontrunning the MEV Crisis*. Blog post. [accessed 26-May-2022]. 2020. URL: https://medium.com/flashbots/frontrunning-the-mev-crisis-40629a613752.

[39] I. G. A. Pernice and B. Scott. "Cryptocurrency". In: *Internet Policy Review* 10 (2 2021). DOI: 10.14763/2021.2.1561.

[40] H. Poston. *Integer overflow and underflow vulnerabilities*. Blog post. [accessed 03-February-2022]. 2020. URL: https://resources.infosecinstitute.com/topic/integer-overflow-and-underflow-vulnerabilities/.

[41] P. Qian, Z. Liu, Q. He, R. Zimmermann, and X. Wang. "Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models". In: *IEEE Access* 8 (2020), pp. 19685–19695. DOI: 10.1109/ACCESS.2020.2969429.

[42] N. F. Samreen and M. H. Alalfi. “SmartScan: An approach to detect Denial of Service Vulnerability in Ethereum Smart Contracts”. In: *Proceedings - 2021 IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB 2021*. 2021, pp. 17–26. DOI: 10.1109/WETSEB52558.2021.00010.

[43] S. Sayeed, H. Marco-Gisbert, and T. Caira. “Smart Contract: Attacks and Protections”. In: *IEEE Access* 8 (2020), pp. 24416–24427. DOI: 10.1109/ACCESS.2020.2970495.

[44] A. T. Sherman, F. Javani, H. Zhang, and E. Golaszewski. “On the Origins and Variations of Blockchain Technologies”. In: *IEEE Security & Privacy* 17 (1 2019), pp. 72–77. DOI: 10.1109/MSEC.2019.2893730.

[45] Solidity. *Solidity documentation*. Programming language. [accessed 26-May-2022]. URL: https://docs.soliditylang.org/en/latest/.

[46] SWC Registry. *Smart Contract Weakness Classification and Test Cases*. Web page. [accessed 26-May-2022]. URL: https://swcregistry.io/.

[47] J. Tidy. *Fear and excitement in El Salvador as Bitcoin becomes legal tender*. News article. [accessed 20-March-2022]. 2021. URL: https://www.bbc.com/news/technology-58473260.

[48] M. Van Der Wijden. *Backrunning in DeFI*. Blog post. [accessed 23-February-2022]. 2020. URL: https://medium.com/@m.vanderwijden1/backrunning-in-defi-301f3cade30a.

[49] Z. Wan, X. Xia, D. Lo, J. Chen, X. Luo, and X. Yang. “Smart Contract Security: A Practitioners’ Perspective: The Artifact of a Paper Accepted in the 43rd IEEE/ACM International Conference on Software Engineering (ICSE 2021)”. In: *Proceedings - International Conference on Software Engineering*. 2021, pp. 227–228. DOI: 10.1109/ICSE-Companion52605.2021.00104.

[50] D. Wang, S. Wu, Z. Lin, L. Wu, X. Yuan, Y. Zhou, H. Wang, and K. Ren. “Towards a first step to understand flash loan and its applications in defi ecosystem”. In: *SBC 2021 - Proceedings of the 9th International Workshop on Security in Blockchain and Cloud Computing, co-located with ASIA CCS 2021*. 2021, pp. 23–28. DOI: 10.1145/3457977.3460301.

[51] I. Yalovoy. *Why Fomo3d 10,469 ETH Block Stuffing Attack Is Important*. Blog post. [accessed 26-May-2022]. 2019. URL: https://ylv.io/why-fomo3d-block-stuffing-attack-is-important/.

[52] J. Yli-Huumo, D. Ko, S. Choi, S. Park, and K. Smolander. “Where is current research on Blockchain technology? - A systematic review”. In: *PLoS ONE* 11 (10 2016). DOI: 10.1371/journal.pone.0163477.

[53] Y. Yuan and F. Wang. “Blockchain: The state of the art and future trends”. In: *Zidonghua Xuebao/Acta Automatica Sinica* 42 (4 2016), pp. 481–494. DOI: 10.16383/j.aas.2016.c160158.

[54] K. Zipfel. *Exploring Commit-Reveal Schemes on Ethereum*. Blog post. [accessed 26-May-2022]. 2020. URL: https://medium.com/swlh/exploring-commit-reveal-schemes-on-ethereum-c4ff5a777db8.