DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Information Systems

Identification of Programming Patterns in Solidity

Franz Sebastian Volland

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Information Systems

Identification of Programming Patterns in Solidity

Identifizierung von Programmiermustern in Solidity

Author:	Franz Sebastian Volland
Supervisor:	Professor Dr. Florian Matthes
Advisor:	Ulrich Gallersdörfer, M.Sc.
Submission Date:	June 15th, 2018

I confirm that this master's thesis in information systems is my own work and I have documented all sources and material used.

Munich, June 15th, 2018 Franz Sebastian Volland

Acknowledgments

Foremost, I would like to thank my thesis advisor Ulrich Gallersdörfer. His office was always open whenever I had any questions about my research or writing. He consistently allowed this paper to be my own work, but steered me in the right the direction whenever he thought I needed it.

I also want to thank Professor Dr. Florian Matthes for the help in shaping the topic and for the opportunity to write this thesis at his chair for Software Engineering for Business Information Systems (SEBIS).

Furthermore, I would like to thank the blockchain specialist Alexandros Papageorgiou from Inlecom. Without his passionate participation and input, the validation of the patterns could not have been successfully conducted.

I also want to thank my friends, who acted as second readers of this thesis, and I am greatly indebted for their helpful comments and their valuable time.

I would also like to acknowledge the university library of Weihenstephan and its staff for providing the peaceful and inspiring atmosphere that made writing this thesis such a great experience.

Finally, I must express my very profound gratitude to my parents and to my girlfriend for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them. Thank you.

Abstract

Blockchains like Bitcoin have become mainstream over the last years and are featured in newspapers and prime time television. With the rise of one of these blockchains, Ethereum, smart contracts are receiving increased attention in business applications and research communities. These immutable pieces of code, living on the blockchain, allow parties to manifest contract terms in the form of program code, without having to trust each other, or the need for a trusted third party. Smart contracts are defined with the help of the programming language Solidity, a smart contract oriented language developed specifically for Ethereum. However, the peculiarities and missing experience with this new language have already lead to the loss of a substantial amount of funds, due to attacks and bugs.

We identify programming patterns for the smart contract programing language Solidity and condense them into a structured catalog that can help beginners, as well as experienced developers, write better and more secure smart contracts. Additionally, research on the usage of already existing best practices is conducted, in order to gain insight over usage patterns and the usefulness of works like this. For this task we present a method that allows the efficient investigation of pattern usage on smart contract byte code, with the help of function identifiers.

This thesis provides an overview over the blockchain technology and smart contracts, presents their problems and risks, and explains how programing patterns can be helpful to mitigate common pitfalls. We give an analysis of related work that has already been published on this topic. The main component of this thesis is a catalog of 14 patterns divided into four categories. Each of these patterns is described in great detail, including their forces, their solutions and a code example. We further describe the evaluation process to validate the patterns. Afterwards we investigate the usage of already existing patterns on the example of the Ownable contract by OpenZeppelin and the Oraclize service. It is shown that currently over 20% of new smart contracts are "ownable" and a little less than 1% use Oraclize as a data provider, which suggests that the smart contract community is willing to accept prefabricated solutions and best practices.

Contents

1 Introduction

1.1 Motivation

Blockchain technology is getting more and more into the focus of the public and the academic world. The first use case of a blockchain, the cryptocurrency Bitcoin [1], was able to achieve capital gains of over 2200%[1] just in the year 2017. This success managed to bring blockchains into prime time television and the front pages of newspapers. Additionally, it led to the emergence of several similar blockchain projects, which some of them being basically just copies of Bitcoin, and others introducing new concepts and ideas that benefit from the blockchain as their foundation.

In general, a blockchain can be thought of as a decentralized ledger, which is secured by cryptography and an elaborate mechanism that takes care of consensus between all participants. While the blockchain of Bitcoin uses this ledger to store transaction of its associated currency, other blockchain applications evolved into different directions. The second biggest (in terms of market capitalization) of this alternative blockchains is called Ethereum and allows users to store whole computer programs on its ledger. These programs are also called smart contracts and have the benefit that, contrary to centralized software, their execution cannot be manipulated by the executing party.

Several programming languages have been proposed to help formulate smart contracts. The most commonly used language is Solidity [2]. Solidity is a contract oriented, high-level programming language and was deliberately designed to look similar to JavaScript, in order to facilitate the entry for new developers. Because a lot of software developers are familiar with the syntax of JavaScript, Solidity is very accessible and could quickly build up a large number of users. However, smart contract programming on a blockchain is a whole new paradigm, substantially different to, for example, web development. Completely new features and pitfalls have to be considered, which especially beginners in this field might not be aware of. In addition, Solidity is very permissive. It does not prevent the introduction of common security flaws and vulnerabilities by itself. The combination of a new but permissive programming language and paradigm, which seems familiar to most beginners, and an ecosystem that revolves around the transfer of digital goods, with a strong increase in value, makes for a dangerous environment. As was to be expected, several bugs and attacks, attributed to incorrect formulation of smart contracts with Solidity, have already occurred [3–5] and lead to the loss of several million USD.

[1]Historical prices of Bitcoin over 2017: `https://coinmarketcap.com/currencies/bitcoin/historical-data/?start=20170101&end=20171231`

In order to provide a guideline for writing more secure smart contracts for developers, as well as a reference point for interested users, this thesis features a collection of patterns for the smart contract programming language Solidity. A pattern is a condensed problem with its respective solution, compiled into a format that allows for easy adaptation and reusability. The aim is to gather already existing, but scattered knowledge, enrich it with personal experience and suitable academic insights, and finally compress it into a clear and accessible structure. This way, beginners can check, if common problems are solved in a certain way, whereas experienced users can find verified solutions and methods that they can apply to their problems.

1.2 Research Questions

The following research questions arose during preparation for this thesis and are answered in the subsequent chapters:

Q1 - What are current challenges in smart contract development using Solidity?

In order to formulate any information with the intent to support smart contract developers, it is necessary to understand their pains. It has to be clarified what the most common vulnerabilities are and where they originate from. What were the problems during the implementation phase of a smart contract that lead to the existence of bugs? What are possible solutions to overcome these problems? We investigate common attacks and other pitfalls during smart contract creation and propose several reasons that could be the culprit for different kinds of bugs.

Q2 - Are there any best practices or patterns in smart contract development with Solidity?

A multitude of different smart contracts, including their source code, have been published. Because different developers usually have to tackle the same, or at least similar problems, we answer the question if programmers found suitable techniques to counter commonly occurring problems. Is it possible to identify such reoccurring solutions? Are these solutions shared with other developers, and if they are, which channels of communication and formats are used? We gather several common solutions and best practices and present them in a comprehensive catalog, after researching several different forms of knowledge sharing between developers, as well as first hand research on the code base.

Q3 - How can the identified patterns be structured and categorized?

To really gain an advantage from the identified patterns, it is necessary to present them in an easily accessible and clearly structured format. What would be a good way to condense the gathered information into a form that provides a clear overview over the problem, the forces and the solution, while still being easy to navigate? Furthermore, the whole catalog should be organized in a way that gives away the intention of a pattern, without having to read the complete section. We achieve all this by adapting a well-proven structure for presenting software patterns to the needs of patterns for smart contract creation. The finished patterns are then categorized into selected categories that give an overview over the general objective of each pattern.

Q4 - Is there a way to measure pattern usage in Ethereum smart contracts?

To get an idea about the usefulness of pattern catalogs, like the one presented in this thesis, it would be beneficial to know if and to what extend, patterns proposed to the smart contract development community, are used. As the code of published smart contracts is, in most cases, only accessible in byte form, it is hard to impossible to re-engineer the high level Solidity code, the contract was programmed in. Is there a way to draw conclusions about pattern usage, only on the basis of byte code? Can we develop an efficient method to search for the presence of certain code components in smart contracts? We answer these questions by proposing a method for searching for used patterns in byte code with the help of function identifiers.

Q5 - Are proposed patterns accepted and implemented by smart contract developers?

Are smart contract developers actually using solutions to common problems, proposed by other developers? Can we find out which patterns are the most commonly used ones? How long does it take for a pattern to find its way into multiple other contracts? Questions like these are answered by using the before mentioned method, which can quantify pattern usage and draw a chronological picture of their occurrences. We evaluate and present our results in a short study about the usage of two patterns.

1.3 Approach

The identified research questions from section 1.2 are solved with different approaches. The approach used for the investigation of pattern usage (**Q4**, **Q5**) is detailed in its own section, in section 6.1. The conducted research approach used to answer the remaining questions (**Q1**-**3**) uses methods of grounded theory. Grounded theory is a qualitative research approach that consists of "a logically consistent set of data collection and analytic procedures aimed to develop theory" [6]. This means that one starts with an idea or a question and develops abstract categories to explain and condense one's data in an iterative process.

In our case, we began with our research questions (**Q1**-**3**) and identified the necessary data in order to answer them, as well as to where to find that data. We started data gathering by collecting information from blogs and chat rooms, deployed smart contracts, academic literature and self-conducted tests:

- **Blogs and chat rooms** provide information published by the developers of the blockchain infrastructure and the programming language Solidity. Additionally, smart contract developers exchange information and present relevant findings.
- **Deployed smart contracts** were an important source for hands-on information on current practices in smart contract development. Over 50 currently deployed smart

contracts were manually examined for reoccurring patterns and common mistakes.

- **Academic literature** was used to gain profound knowledge over underlying technology, like blockchains and smart contracts and was used to define the borders to already existing research on that field. The most relevant related work is compiled in chapter 3.

- **Self-conducted tests**, using a smart contract test environment, were used to confirm and test ideas, as well as gather experiences with the pitfalls of smart contract programming.

The collected ideas from these sources were grouped into bigger concepts, which would later evolve into the desired patterns. On the basis of these new concepts, further data could be collected to gather additional, profound information. By this way, more and more information for each concept could be gathered, which allowed us to condense that information into a pattern structure. The final patterns feature only the necessary information with a focus on the most important aspects.

The benefits of using such a grounded theory approach are, amongst others, that research findings are ecologically valid, because they are derived from actually deployed smart contracts. Furthermore, the findings are not closely tied to already existing theory, which allows for a fresh start and a view from a new perspective on a topic.

1.4 Outline

The remainder of this thesis is structured as follows. Chapter 2 covers the necessary fundamentals for this work. This includes the explanation of the terms blockchain and smart contract, an overview of the history of smart contract programming languages, explanation of attacks and other challenges of smart contracts, and concludes by an explanation of patterns and how they are beneficial for software development. Chapter 3 introduces related work. The main part of this thesis can be found in chapter 4, which features the catalog of the identified patterns. Chapter 5 deals with the different ways the presented patterns have been evaluated. The second part of this thesis, the investigation of pattern usage, is covered in chapter 6. This thesis is concluded with chapter 7, where we give an outlook over possible future work and sum up all relevant findings of this paper.

2 Fundamentals

Blockchains and working smart contracts are a relatively new and complex topic. Correspondingly high is the potential for confusion and the amount of conflicting information. The aim of this chapter is to lay out the foundations necessary for understanding the remaining chapters. This includes an introduction to the relevant mechanics and characteristics of a blockchain, the history and technical implementation of smart contracts and a short summary of the evolution of smart contract programming languages. Further, we have a look at the threats to smart contract security, due to attacks and bugs, before explaining the concept of patterns and how they can help develop better software, in particular with Solidity.

2.1 Blockchain

As a short definition, it can be stated that a blockchain is a continuously growing list of records, which are bundled in blocks. These blocks are linked one to another, like a chain, and are secured by cryptography. Over the course of this chapter, we expand this short definition and give an idea of the general concept of a blockchain, on the example of Bitcoin in subsection 2.1.1. Afterwards, in subsection 2.1.2, we show the differences made in the implementation of Ethereum and the reasoning behind them. Because this thesis focuses on the technical implementation of smart contracts rather than the underlying architecture, the following overview is a simplification of the actual technology and focuses on the aspects relevant for the remaining parts of this work. The specified sources provide a profound explanation of blockchains and especially Bitcoin for the interested reader.

2.1.1 Bitcoin

Idea

The idea of Bitcoin was proposed by a person, or a group of people, under the pseudonym Satoshi Nakamoto in 2008 [1], as a purely decentralized version of electronic cash. Nakamoto's attempt was the first to actually solve the double spending problem[1] of digital currencies. A digital currency, like Bitcoin, lets people exchange money over the internet, without the need for any regulators or central authorities. Over the years, multiple similar projects emerged that tried to copy or improve Bitcoin in one way or the other. However, up until the time of writing, Bitcoin remains the most popular

[1]Double spending is a potential flaw in digital currencies that allows one token to be spent multiple times.

cryptocurrency in terms of market capitalization[2]. The underlying technology that made the concept of Bitcoin and its success possible, is called blockchain, the concept of which was also introduced by Nakamoto, together with the proposal of Bitcoin [1]. It provides a framework in which individuals can make peer-to-peer (P2P) transactions without having to put trust into a third party, or even into each other.

Analogy

A blockchain can be imagined like a traditional ledger in a bank, before the invention of computers. People wanting to transfer money would write transfer slips and give them to the banker. In the evening, the banker would collect all the transactions that incurred over the day, validate them, and write them down into the ledger. Once they are written down, the transactions are seen as fixed. Everybody's balance is up to date and the transactions cannot be redone. A blockchain works in a similar way. The users' transactions are bundled in blocks. The job of collecting the transactions and writing them down, is taken over by so called miners. The state after a certain interval is not written down on paper, but a decentralized electronic ledger, which consists of a long chain of blocks.

Distribution and Mining

A transaction in the sense of the Bitcoin blockchain is in most cases a transfer of any denomination of the currency used in the network, which is also called bitcoin, from one address to another. The sender has to sign the transaction, in order to proof that it is in fact him who issued this transfer, before it is broadcast to the network. More accurately, it is broadcast to his peers in the P2P network. The transactions are bundled into a block by special nodes called miners. Miners compete with each other to solve a cryptographic puzzle. The first miner to solve the puzzle appends his block to the blockchain by broadcasting it to the network. This way a new block is "mined" roughly every ten minutes [7]. The incentive for miners to compete in this computationally heavy puzzle is the reward, in the form of Bitcoin, for the winner of the race.

Besides the determination of who appends the block to the chain, the puzzle serves other purposes. It acts as a proof-of-work (PoW) and is necessary to make sure that a certain amount of resources (here: electricity) were used up. This prevents attackers to create an arbitrary large amount of virtual nodes and take over the network.

Block Components

Each of the blocks that make up the blockchain consists of a header and a collection of transactions [8]. The transactions are stored in a hash tree, with each transaction as a leaf and a hash as the root of the tree. The root hash acts as a convenient way to prove the inclusion of a transaction in this block. The header contains some metadata

[2]List of popular cryptocurrencies and their market values: `https://coinmarketcap.com/`

(e.g. the timestamp and information related to the mining puzzle), the root hash of the transaction tree and the hash of the header of the previously mined block. This connection of blocks resembles a chain, therefore the name blockchain. A graphical representation of the structure of a block and the connection between them can be found in Figure 2.1.

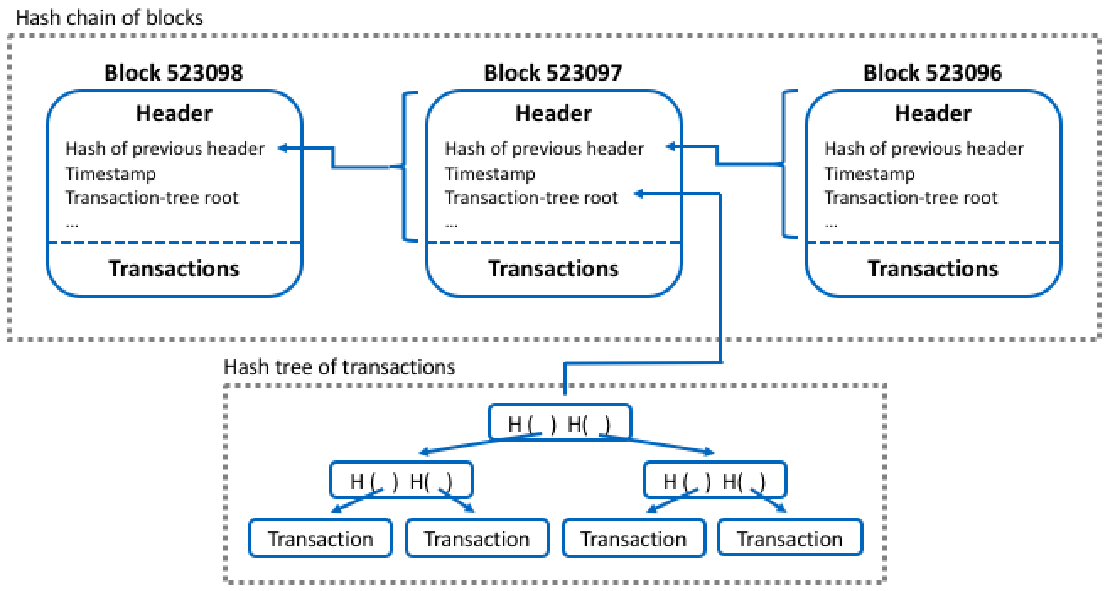

Figure 2.1: Structure of a block, including its header, its transactions and the connection to predecessor and successor. Inspired by [8].

The inclusion of the previous block's header is also a crucial point for providing immutability to the blockchain. If someone were to manipulate a transaction after its inclusion in a block, the hash of the root of the transaction tree would change, once it is recalculated. This would led to a change in the hash of the block's header, as the root hash is a part of it. That in turn would mean that the block hash stored in the header of the successor would not match anymore, resulting in the chain being no longer intact. Such a change would not be accepted by the network, making a subsequent change in existing blocks almost impossible.

Network and Consensus

A copy of the blockchain, including all transactions that ever occurred, is stored locally at every actively participating node of the network. Nodes that maintain a copy of the blockchain are called *full nodes*. Because a copy of the blockchain can be several hundred gigabytes big, there is also the option to participate as a *light node*. Such nodes do not posses a full copy of the blockchain. They rather download only the block headers for validation. Using this method, devices with limited storage capacity, like smartphones, can access the network.

To ensure that there is common agreement on what the current state of the blockchain is, despite of all the asynchronous transactions, Nakamoto proposed a decentralized consensus mechanism. The desired consensus is achieved by four independent processes occurring across the network [7]:

- Independent verification of every transaction by each full node.
- Independent aggregation of those transaction into blocks by miners, providing PoW.
- Independent verification of every new block by every node.
- Independent selection of the chain with the most computation done through PoW by every node.

This consensus mechanism paired with the securities of cryptography and the proof-of-work lead to a system, which allows users all over the globe to transfer digital money without the need for a central authority.

2.1.2 Ethereum

Idea

Ethereum is a blockchain system, just like Bitcoin. However, it is running a different protocol and follows slightly different rules. While Bitcoin is primarily used for sending money from one address to another, Ethereum can be thought of as one big computer, consisting of several small computers around the world.

The idea of Ethereum was introduced by Buterin [9] in 2014, with the intent to "create an alternative protocol for building decentralized applications" (more on decentralized applications in subsection 2.2.2). In essence, the Ethereum blockchain is a transaction-based state machine, similar to Bitcoin. It is in a certain state, when it is given some inputs via transactions. Based on these inputs, the system transitions into a new state. Most of the other concepts introduced with Bitcoin also hold true: transactions are being grouped into blocks by miners, while being individually validated by participating nodes to form an ever-growing chain of blocks.

Accounts

One essential difference to Bitcoin in Ethereum is the notion of accounts. In Bitcoin, an account is basically just a collection of different addresses and their associated private keys. In Ethereum an account only has one address and a state associated with it. There are two different kinds of accounts in Ethereum [9]:

- **Externally owned accounts**: are controlled by a private key. Also referred to as user accounts.

- **Contract accounts**: are controlled by the code they are associated with. Also called smart contracts (more about smart contracts in section 2.2).

Contract accounts are one of the features that make Ethereum so special. They are able to embody code on the blockchain and do not have a corresponding private key. Externally owned accounts can send messages to other external accounts, or contract accounts in the form of transactions. Transactions between two externally owned accounts are most of the times simple value transfers, with the difference that in Ethereum the underlying currency is called *Ether* (ETH). A transaction from an externally owned account to a contract account can be used to invoke functions at the contract.

Contract accounts have no corresponding private key. They exist autonomously on the blockchain and are not able to initiate any behavior on their own. They are thus reliant on being triggered by externally owned accounts or other contracts. This also means that any action that happens on the Ethereum blockchain is always initialized by a transaction from an externally owned account.

Ethereum Virtual Machine

Another crucial extension is the Ethereum Virtual Machine (EVM). It is a quasi-Turing complete[3] virtual machine, specific to Ethereum that processes every transaction, and its possible executions of code, in a trust-less environment. Every participating node in the network runs their own instance of the EVM.

Transaction Fees

The EVM is only quasi-Turing complete, because it is limited by transaction costs [10]. A Turing complete systems needs some sort of conditional repetition, like loops. Such instructions can potentially run indefinitely long. In a decentralized system, where every instruction has to be computed by each participating node, it is therefore necessary to limit execution time and scope. In Ethereum this is done by charging a fee for each computational step. A financial deterrence can help avoiding executions that are too computationally heavy, as well as prevent infinity loops that would congest the network.

Transaction fees are calculated in units of *gas* [11]. Each operation on the network, including sending transactions and executing operations with the EVM, has a certain gas cost associated with it. The more computationally heavy the execution of a transaction is (e.g. storing a lot of data in the storage of a contract account), the more gas has to be payed as a fee to the miner. Gas is payed for in Ether. As with Bitcoins, Ether can be split into smaller units. The smallest subdenomination of Ether, and also the one in which all integer values of the currency are counted in, is Wei[4] [11].

[3]Turing completeness is achieved by an instruction set, if it can be used to solve any computational problem.

[4]$1 Wei = 10^{-18} Ether$

Consensus and Mining

Consensus and mining in Ethereum work similar to the way in Bitcoin, with some differences. One significant variation is the block time. Instead of every ten minutes, a block is appended to the blockchain roughly every 15 seconds. A faster block time for Ethereum makes sense, as its use case is different to the one of Bitcoin and interactions with the blockchain need to be in shorter intervals, when dealing with complex applications. However, this shorter block time increases the chances for independent miners to produce the same block. In that case, normally one of them would end up with no reward and having wasted a lot of computational power without aiding the stability of the network. A modified version of the Greedy Heaviest Observed Subtree (GHOST) protocol [11] is used to mitigate this issue. GHOST makes use of these otherwise wasted blocks, by rewarding them with ether and including them into the consensus algorithm.

2.2 Smart Contracts

This chapter dives deeper into one of the reasons Ethereum was developed for in the first place: smart contracts. In subsection 2.2.1, we provide the background for this new technology, as it exists today, by reviewing the basic idea of smart contracts, formulated by Szabo [12]. Afterwards, subsection 2.2.2 describes the technical implementation of this idea in Ethereum.

2.2.1 Basic Idea

The idea of smart contracts is not new. The term was coined by Szabo as far back as in 1994, as a "computerized transaction protocol that executes the terms of a contract" [13]. In a later publication Szabo refined his definition of smart contracts:

> A smart contract is a set of promises, specified in digital form, including protocols within which the parties perform on these promises [12].

The general idea behind this concept was to embed common contractual clauses, like liens, in hard- and software that would make the breach of a contract expensive for the breacher. Following this idea, traditional contracts can be encoded in computer programs that take over the task of executing the contract. This form of commitment has several advantages over a traditional contract: lower costs, easier to audit, clear interpretation due to less subjectivity. Additionally, it is now possible for machines to easily enter into legal agreements.

Szabo gives the example of a vending machine as a real-life example for a primitive ancestor to a smart contract [12]. It takes in coins and delivers change and a product in return. All of this is done in an automated way like a finite state machine.

In the real world, contracts regulate the agreement between two or more parties that do not necessarily trust each other. In order to overcome the issue of missing trust, a third party (e.g. a notary) is needed to enforce the contract. Therefore, the problem of enforcement is shifted from the initially participating parties to the third party. How can this behavior be transferred to a smart contract executed via a computer program? Who would execute the program? If only one party executes it, the remaining parties would trust the executing one to provide the correct results. If all parties would execute the program, there would be a problem, if the individual results would differ from each other. Therefore, a trusted third party is needed as well, in order to execute the program on behalf of the other parties. Szabo envisioned a "trustworthy virtual computer" using "post-unforgeable transaction logs" and "mutually confidential computation" to take over this task [14].

Even though trust-less systems, using cryptography, were known at the time, they were not practically available for an implementation [15]. Only with the uprising of blockchain technology, the revive of Szabo's idea of smart contracts became possible. From here on, the terms *smart contract* and *contract* are used interchangeably, because we will put our focus on virtual contracts deployed on blockchains.

2.2.2 Technical Implementation in Ethereum

Blockchain technology, with its promise to provide consensus on a trustworthy and immutable ledger, could be the perfect host for smart contracts. The Bitcoin blockchain offers a rudimentary, stack-based scripting language that allows for some use cases (e.g. escrow transactions) but lacks the required expressiveness to implement even simple smart contracts [8]. Ethereum, with its virtually Turing complete EVM, is the first incarnation of a blockchain, which features a programming language that is able to express even complicated smart contracts.

The technical implementation of the idea of a smart contract envisioned by Szabo is a computer program that runs on the blockchain, i.e. is executed by all nodes taking part in the consensus protocol. In the specific case of the Ethereum blockchain, such a contract consists of its program code, its data storage and an account balance. A smart contract can be created by every user, and even other contracts, by sending a transaction containing the contract code to the address `null`. Once this transaction is mined and therefore stored on the blockchain, the smart contract is fixed and cannot be changed anymore. This immutability is a key feature for smart contracts, as it removes the need for trust in other parties. However, it also removes the possibility to upgrade contracts or fix bugs in a conventional way, which can be bypassed by using the upgradeability patterns from section 4.3.

The specified logic of a contract is executed by every full node, as part of the consensus algorithm. The nodes have to verify that the state of the blockchain is correct after every interaction with a contract. The consensus protocol makes sure that the nodes come to an agreement on the result of the contract execution. The entry points of smart contracts are functions. They are executed when the contract receives a transaction from a user or a call from another contract. Users can invoke a function, by sending a transaction, containing the identifier[5] of the function and any desired function parameters to the contract's address.

During the execution of its code, a contract can read or write data from/to its persistent storage. Additionally, memory can be used for data processing, but will be wiped clean after the function call finished. Smart contracts are also able to receive money in the form of Ether attached to a transaction, or transferred from another contract. The obtained funds are stored in the account's balance and can in return be sent to other users and contracts. Conceptually, smart contracts in Ethereum take over the envisioned role as a trusted third party. It is trusted for correctness and availability, however not for privacy [16]. The entire state of a contract and every transaction are stored on the blockchain and visible to the public.

Smart contracts do not necessarily stand on their own but can be a part of an application, in order to make it decentralized. These decentralized applications (**DApps**) are applications that run on a decentralized network, like Ethereum, and often use a smart contract for the execution of logic and the storage of data. To facilitate the interaction with the contract and provide a more appealing user experience, oftentimes a web front end is used. With the help of libraries like web3.js, functions of a contract can be called with the click of a button. This ease of operation could open the door for users unfamiliar with blockchain technology and contribute to the success of smart contracts and Dapps in the future.

2.3 Smart Contract Programming Languages

Even though Ethereum and the EVM have only been existing since 2014 [9], there has already been quite some development in regards to programming languages for smart contracts on the platform. Developers wanting to implement smart contracts can choose between several languages with different backgrounds and satisfying different needs. The most prominent ones being LLL, Serpent, Mutan and Solidity. Some promising languages have also been only partially released and are still under heavy development like Viper and Bamboo. To provide a proper background about certain design decisions in Solidity and to give the right context for its position in the smart contract programming language ecosystem, this chapter will give a brief summary about the languages mentioned above.

2.3.1 The Beginnings

This section features the first generation of smart contract programming languages, the languages used before Solidity.

[5]An 8 byte long set of characters obtained by hashing the function name with the data types of its input parameters.

LLL

At the beginning of Ethereum, there was only the turing complete EVM that could be used to implement smart contracts. Developers, nonetheless, were interested in creating smart contracts, but were lacking the tools to bring their ideas on the blockchain. The Ethereum developers knew that they could not come up with a high-level language in a decent amount of time. However, it was necessary to provide something to the developers out there, before the initial interest in Ethereum faded. Therefore, LLL was created. LLL stands for *Low-level Lisp-like Language* and is, as the name suggests, similar to Lisp [17]. According to Vitalik Buterin, the conceptual inventor of Ethereum, "LLL was always meant to be very simple and minimalistic; essentially just a tiny wrapper over coding in ASM[6] directly" [18]. Nowadays, LLL plays little role in the Ethereum community. The last topic created in the LLL subsection of the Ethereum Community Forum[7] was created in April 2015 which suggests that LLL is barely used anymore.

Serpent

Serpent is another one of the early programming languages for smart contracts and is intentionally "designed to be very similar to Python" [19]. Sources differ as to if it is a low- or high-level language [19, 20]. Nevertheless, the description of it being a low-level language with various high-level features, could be seen as a compromise. This combination allows for an easy to use programming language with special features for contract programming, while still having the efficiency benefits of a low-level language [19]. Serpent has access to opcodes, as it is a superset of LLL, and also uses LLL to compile into bytecode [18]. Until Solidity (see subsection 2.3.2) was released, Serpent was considered the flagship of the three existing languages LLL, Serpent and Mutan. In 2017, Augur, one of the most popular and early crypto tokens and written in Serpent, commissioned Zeppelin Solutions to audit the Serpent compiler. Their results included several critical severities that had been undetected up until then [21]. Shortly before the publication of the audit results, a statement was published which declared that Serpent is seen as deprecated [22].

Mutan

Based on Go, Mutan is the third of the first generation smart contract programming languages, besides LLL and Serpent. It is similar to C. Variable types have to be known at compile time, making it statically typed. Out of the three, it is considered the only true high-level language and it compiles directly to the native Ethereum assembler. As of March 2015, Mutan is seen as deprecated [23].

[6]ASM refers to the Ethereum assembly language, a low level programming language, with a high correspondence between the language and the EVM's machine code instructions.

[7]LLL section of Ethereum Community Forum: `https://forum.ethereum.org/categories/lll`

2.3.2 The Standard: Solidity

The high-level language Solidity was introduced to developers in August 2014 [24]. Looking at its syntax, similarities to JavaScript and C++ can be discovered. The decision to give it a similar look as the famous scripting language JavaScript, was made to make it appealing to a broad mass. As it is supposed to be used for contract creation, several features are tailored towards that purpose (e.g. the concept of contracts instead of classes). Solidity's programming paradigm is described as contract oriented. The finished contract is compiled to assembly where it gets optimized before it is compiled to bytecode for the EVM. Solidity was first developed to overcome several problems and limitations of the other languages available at that time (e.g. introduction of functions, or the connection to JavaScript API for DApps). After its introduction, it was speculated that "Solidity could one day very well replace all three existing high level languages (Mutan, Serpent and LLL)" [25]. This conjecture turned out to be true and as of the time of writing, basically every published smart contract is written in Solidity. One reason for this success probably lies in the support from the Ethereum Foundation, who embraces Solidity as the go-to language [26]. However, there have been also critical voices about Solidity, which denounce its permissiveness and its unpredictability [27].

2.3.3 Upcoming Languages

This section provides an outlook on two currently developed smart contract programming languages and their intentions. There are several similar projects under current development, which will not be discussed in this work.

Vyper

One of the most recently introduced high-level smart contract programming languages is Vyper. Developed by Vitalik Buterin and first published in November 2016, Vyper is striving to improve some of the shortcomings of Solidity. The general idea behind it is to restrict certain delicate aspects, in order to make the language more straightforward and easier to audit. Vyper can be seen as the successor of Serpent, and it is also derived from Python. According to its developer, it "is not meant to replace Serpent, Solidity and LLL" rather than "to provide different properties and tradeoffs compared to existing languages" [28]. Vyper tries to tackle the problem of bad readability of some contracts, especially ones deliberately being written to mislead unsuspecting users, by removing potentially confusing features like inline assembly and modifiers [29]. "Viper is strictly less powerful than Serpent in certain intended ways, such as being decidable and thus not having dynamic-length loops" [28]. All these changes have not only been made to make life easier for programmers, but also with the people in mind who have to audit code, as it is now "maximally difficult to write misleading code" [29]. As of the time of writing, Vyper is still in an alpha stage and under development.

Bamboo

Similar to the motivation behind Vyper, the intention of Bamboo is to improve readability in smart contracts and reduce surprises of any sorts [30]. Development began in September 2016 and the first compiled bytecode was produced by June 2017 [31]. The high-level language promises some characteristics similar to Vyper, for example the lack of loop constructs (Vyper only got rid of dynamic length loops). The development of Bamboo is still in a very early stage and not as active[8] as the one on the Vyper project.

2.4 Attacks and Other Challenges

Ethereum and complex smart contracts are still relatively new. Their security landscape is constantly changing, as new security issues and bugs are discovered. The combination of these two factors - new system and changing security issues - make it a difficult undertaking for developers to design secure smart contracts. Several real-world examples show that it is not always possible to avoid all mistakes and pitfalls. A list compiled by Vitalik Buterin indicates that even in the first years of Ethereum, several incidents lead to the loss of substantial amounts of money [32]. Examining this list, the listed bugs can be divided into three groups: **attacks**, **syntactic bugs** and **conceptual problems**. In some way, these categories are interlinked, as for example a syntactic bug causes a vulnerability that can be abused by an attack. However, this chapter tries to differentiate and give a short overview over possible attack vectors, bugs and conceptual peculiarities that developers should be aware of. It further explains, how the proposed patterns in chapter 4 can help attenuate these issues.

2.4.1 Attacks

Several attacks on smart contracts in the past, like the ones on the DAO [3] or the Parity wallet [5], show that vulnerabilities are actively searched for and abused on Ethereum. Analysis on currently active smart contracts by Nikolic et al. [33] revealed that there are at least 34.000 vulnerable contracts deployed on the blockchain, which all have at least one security issue that can be exploited. The corresponding exploits have been researched by Atzei et al. [34], who present 12 attack vectors that can target smart contracts at three different levels of execution: in Solidity, the EVM and the blockchain itself. Another good overview of vulnerabilities is given in [35], where 10 security issues are explained in detail and are sorted by their total loss in ether, they are estimated to have caused. Number one in this list, due to the high amount of damage in the DAO exploit, is the attack vector called re-entrancy. Because the aim of some of the presented patterns is to avoid this attack, this example is explained in more detail in the following:

The re-entrancy attack is carried out by manipulating the control flow in a way that was not intended by the smart contract developer. The following function contains a

[8]59 commits from 10 contributers for Bamboo vs. 265 commits by 57 contributers for Vyper in the first half of 2018

potential vulnerability allowing an attack to re-enter the contract:

```
1  function withdraw(uint _amount) {
2    require(balances[msg.sender] >= _amount);
3    msg.sender.call.value(_amount)();
4    balances[msg.sender] -= _amount;
5  }
```

Listing 2.1: Re-entrancy attack example

In a normal withdrawal process, a user would call the function with a specified `_amount` to be withdrawn. Line 2 makes sure that the calling user has enough credit to request the withdrawal. If this is the case, the amount is transferred in line 3, before the balances are reduced in line 4.

This process can be exploited, if an attacker implements a malicious contract containing a special fallback function[9]. This fallback function is triggered by the ether transfer in line 3 and takes over control flow. It is then possible to issue a new call of the `withdraw(..)` method in the fallback function, before the reduction of the balance in line 4 has happened. Because the balance has not been reduced yet, the check in line 2 would still pass and the amount of ether would be transferred to the attacker once more. This would trigger the malicious fallback function once again. This iterative process would continue, until a point specified by the attacker, or the transaction runs out of gas.

This vulnerability is a good example for how easy it is to write code that can be exploited. Several academic works try to deal with this problem by analyzing smart contracts for vulnerabilities. Either via live monitoring [36], analysis tools [37] or operational semantics [38]. However, all of these approaches only deal with the end result and therefore, do not tackle the actual cause of the problem. The proposed patterns in this thesis can potentially give insights into common pitfalls and therefore help avoiding vulnerabilities and make contracts more secure from the start.

2.4.2 Syntactic Bugs

Solidity is a relatively new programming language and smart contract programming in general requires a different engineering mindset, which most of the new developers coming into this field are not used to. Because of these reasons it is not surprising that mistakes while programming happen. Research shows that over 12,000 ether are already lost due to typing errors in addresses [39]. While these lost funds are not directly related to bugs in smart contracts, it still shows that even in a context with financial implications, people are making simple mistakes.

A case where such a bug in a deployed smart contract can be observed is Rubixi[10]. A last minute change in the name of the contract lead to a mismatch between the contract name and the name of the constructor function. Solidity treated the alleged constructor, which would only be executed once at contract creation, as a regular function. This

[9] A function without name, which is called in case no function identifier matches the called function.

[10] Rubixi contract: `https://etherscan.io/address/0xe82719202e5965Cf5D9B6673B7503a3b92DE20be#code`

made it possible for everyone to become the owner of the contract and pay out all of the collected fees.

Cases like Rubixi show that even a small mistake, not caught by the compiler, can have fatal results. Patterns cannot prevent all of such errors, but it can reduce the chances of them happening by providing a variety of reusable pieces of code.

2.4.3 Conceptual Problems

Research shows that even the implementation of a relatively simple contract is non trivial for beginners in smart contract programming [16]. One of most commonly problems faced by developers new to smart contracts are of conceptual nature, like understanding the EVM as a state machine, or not knowing of the public visibility of transactions and storage on the blockchain. Failures in grasping or ignoring the concepts of a blockchain can lead to unintended behavior and severe vulnerabilities.

The fact that transaction inputs are visible to everyone, became a fatality for a rock-paper-scissors game on Ethereum [40]. Everyone monitoring incoming transactions at the contract was able to see, which move the first player made and could act accordingly, in order to win the game. A critical flaw that can be avoided by using cryptography and a commit-reveal schema.

Patterns, like the ones proposed in chapter 4, can help to provide a better idea of the concepts of a blockchain by explaining them in the context of certain use cases.

2.5 Patterns

The original definition of patterns was made by Christopher Alexander et al. and defines it as a "three-part rule which expresses a relation between a certain context, a problem, and a solution" [41]. In a later publication, Alexander describes the purpose of a pattern in the following way:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [42].

Even though these definitions were given over thirty years ago and referred to patterns in the context of architecture and urban planning, their implications still hold true until today and can be applied to many other fields, including software development.

Almost 20 years later, in 1994, Gamma et al. published their book about design patterns in software engineering [43]. They described and documented 23 design patterns and assigned them into three categories. Buschmann et al. introduced a way to distinguish different kinds of patterns [44], depending on their radius of influence:

- **Architectural Pattern:** Provides a general idea of the structural composition of software systems. It includes predefined subsystems, and specifies their responsibilities and the way they interact with each other.
- **Design Pattern:** Provides a general blueprint for a particular subsystem in a software system. It tackles commonly occurring problems by providing a customizable solution. It often solves problems in the context of communication between components.
- **Coding Pattern/Idiom:** Describes how to implement certain aspects of components or how they work together. It uses the distinct features of a language and therefore is language-specific and low-level.

In this work we want to put focus on solutions for the contract oriented programming language Solidity. Therefore, the patterns presented later on are mostly from the last group of patterns, the low-level idioms. Some patterns, however, describe the communication between components and could also be assigned to the group of design patterns. A clear separation between idioms and design patterns is not always possible.

One of the ideas behind the creation of patterns by Alexander was that ordinary people, not only professionals, should be able to design for themselves and their community. In the context of a programming language like Solidity, developed with the intention to design smart contracts, this idea still holds true: smart contracts are eventually implemented by software developers, but to reach a critical mass for adoption it will be necessary that not only professionals, but also ordinary people can interact with them. Out of 7.5 billion people on earth, only 22 million are software developers[11] and therefore could have an interest in learning an emerging language like Solidity. If smart contracts are supposed to be used by a broader user group, it will become important that people who have no prior Solidity experience (lawyers, accountants, etc.) are able to understand smart contracts. It could therefore be argued, that Solidity is not only a programming language for developers, but also for users who want to know what they are getting into, before interacting with a smart contract. This is an immense challenge. Most people already find it hard understanding legal texts in their native language, let alone in a programming language they are not familiar with. Solidity patterns could be a source for providing trusted concepts for certain scenarios, including explanations that facilitate the entry into this new territory.

Also experienced programmers can benefit from the use of patterns. The reuse of successful and tested software architecture can reduce flaws and security gaps. This is especially usefull for smart contracts, since they often handle substantial amounts of value and errors are easy to exploit, while hard to fix [45].

Patterns can be a great help for developers as well as users. By providing a set of common and often occurring problems with their respective pitfalls and solutions, the

[11]EDC Global Developer and Demographic Study 2017: `https://evansdata.com/reports/viewRelease.php?reportID=9`

implementation of new smart contracts can be facilitated. They can also help to audit and understand existing contracts.

The remaining section focuses on the structure the patterns are presented in and the categories into which they can be classified.

2.5.1 Pattern Structure

In order to gain a tangible benefit from the use of patterns it is necessary to know about the reasons, decisions and alternatives that led to the final form of a pattern. Understanding the background of a pattern is one step for a proper reuse of the solution and adaption for a particular problem. As shown in the related work in chapter 3, other pattern catalogs for Solidity limit themselves on code and only provide minimal explanation, or even none at all. To facilitate the understanding of the solution and to provide a broad overview over the problem as well as the solution domain, this work lays the focus on the holistic view of the pattern. A code example is provided as well, but should only be regarded as a reference point of how an implementation could look like.

To get a clear overview over the patterns and make them easy to remember and compare, we are using a consistent format for the description of the patterns. They are divided into different subsections, which have been inspired by the renown catalog of design patterns by Gamma et al. [43]. Adjustments have been made to account for the unique properties of smart contracts. A comparison between the structure presented by Gamma and the one used in this work is presented in Table 2.1.

Table 2.1: A comparison of the structures for pattern presentation.

No.	Structure used by Gamma et al.	Structure used in this work
1	Pattern Name and Classification	Pattern Name and Classification
2	Intent	Intent
3	Also Known As	
4	Motivation	Motivation
5	Applicability	Applicability
6	Structure	
7	Participants	Participants & Collaborations
8	Collaborations	
9	Consequences	
10	Implementation	Implementation
11	Sample Code	Sample Code
12		Gas Analysis
13		Consequences
14	Known Uses	Known Uses
15	Related Patterns	

It can be seen that some sections have been omitted, while others have been added or rearranged. The reasoning for the changes, as well as an explanation of the sections is given in the following:

1. The *name* of the pattern should be easy to remember and should convey the essence of the pattern. The pattern *category* provides the general background of the pattern as introduced in subsection 2.5.2.
2. The *intent* gets to the point of the pattern and delivers its purpose in one short sentence.
3. The subsection for different names has been omitted in this work, due to a lack of alternative names.
4. In the *motivation* subsection, the problem to be solved is introduced, as well as additional background that played a role in the formation of the pattern.
5. In *applicability*, several situations are presented, in which the pattern can be applied.
6. The structure in Gamma et al. features a graphical representation of the classes. Nearly all patterns presented in this work consist of only one contract, therefore this subsection is omitted. The structure of patterns including several contracts is made clear in the implementation section.
7. The *participants* are the subjects participating in the pattern. Because we observed a lower amount of participants than in the reference literature, we decided to combine the participants and their collaborations into one subsection.
8. The *collaborations* are covered together with the participants.
9. The consequences have been moved after the implementation section, to make its contents better understandable for the reader.
10. In the *implementation* part, a detailed explanation over the used techniques and particularities of the pattern is given.
11. The *sample code* usually consists of an exemplary contract, which is either a generic version or a special use case.
12. The *gas analysis* section is only present in patterns of the economic section. It conducts an empiric analysis of how much gas can be saved by its usage.
13. The *consequences* include the implications, as well as trade-offs that should be considered when implementing the pattern.
14. In the *known uses* section, usually two examples of real world applications of the pattern on the blockchain are given.

15. The related patterns subsection has been omitted due to the lack of relations. In case a pattern is used in other patterns, this is noted in the corresponding subsection.

2.5.2 Pattern Categories

To introduce a certain degree of order in the pattern catalog, the patterns have been clustered into four categories, depending on their purpose. These categories form families of related patterns. They can facilitate the search for a suitable pattern and keeping track over the selection. The four selected categories that are best to describe the purpose of the patterns, are:

- **Behavioral:** Patterns that influence or extend the behavior of a smart contract.
- **Security:** Patterns that mitigate common attack vectors or add security functions to a smart contract.
- **Upgradeability:** Patterns that help a smart contract being upgraded or maintained.
- **Economical:** Patterns that can reduce the gas required when interacting with a smart contract.

The proposed categories are not clearly distinct and some patterns have attributes that would also make them fit into a different category to some extend. However, they have been assigned to the family, they fit the most.

3 Related Work

The goal of this chapter is to give an overview over other attempts to solve the problem of identifying programming patterns around smart contracts and highlighting the work done by other authors that ties in with this thesis. It gives a context to the inspirations that shaped this work, as well as demarcates it from other publications, by showing differences and the decisions that lead to them. This chapter is divided into two sections, with the publications in each section sorted by the publication date or the creation of the repository, starting with the oldest:

1. **Scientific Contributions**: a presentation of the most important scientific papers that deal with smart contract patterns.
2. **Published Pattern Catalogs**: a compilation of already published Solidity pattern catalogs and their features.

Table 3.1 at the end of this chapter, gives a compact overview of the related work.

3.1 Scientific Contributions

One of the first publications writing about pitfalls in smart contract programming is, *Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab*, from Delmolino et al. [16]. This paper documents the experiences in teaching smart contract programming to undergraduate students in a lab course. The authors present typical pitfalls made by beginners when designing smart contracts and suggest best practices to avoid them. The discovered errors include: erroneous state machines, sending sensitive data in cleartext, misaligned incentives and Ethereum specific mistakes. Based on the experiences during the lab, online open course materials for smart contract programming were released. While this paper revolves around smart contract programming with Serpent and not Solidity, the insights in common mistakes and the mindset of beginners in this field is valuable.

In *An empirical analysis of smart contracts: platforms, applications, and design patterns* [46], Bartoletti and Pompianu propose a taxonomy of smart contracts on Bitcoin and Ethereum into five categories. They quantify a sample of 811 smart contracts into the proposed categories via manual inspection. They further identify nine common design patterns during their inspection and quantify their usage, again via manual inspection. This paper provides valuable insights into what the most common use cases for smart contracts are. Additionally, it is the first attempt to investigate patterns and their usage in the context of smart contracts. The presented patterns are described very briefly and without any code samples.

The application of design patterns in a health care context is described by Zhang et al. Their paper, *Applying Software Patterns to Address Interoperability in Blockchain-based Healthcare Apps* [47], describes how the use of four already existing design patterns can help resolve common interoperability challenges in blockchain applications. The described patterns - Abstract Factory, Flyweight, Proxy and Publisher-Subscriber - are implemented in the context of a blockchain based health care application. The four patterns are presented in a clear structure, including sample code in Solidity. The work does not introduce any new Solidity or blockchain specific practices. The reuse of already existing software design patterns in the context of a blockchain, however, gives valuable insights in how other existing patterns could be reapplied in smart contract programming.

In *Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach* [48], Mavridou and Laszka present a framework for designing smart contracts as finite state machines. They implement a tool for automatically generating Ethereum contracts with Solidity. Additionally, four programming patterns are implemented as plugins for the tool, in order to enhance the security and functionality of the created contracts. The patterns implemented as plugins are Locking, Transition Counter, Automatic Timed Transactions and Access Control. This work is an interesting take on automatic smart contract creation and the possibility to use patterns as building blocks in this process. The used patterns are described briefly and include sample code.

The first scientific catalog of patterns was published by Wöhrer and Zdun in their paper, *Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity* [49]. Their work includes the elaboration of six smart contract patterns, described on the basis of Solidity. These patterns are all supposed to take care of making smart contracts more secure and are presented with a short description and sample code. This publication features only patterns from one category, namely security. However, an associated GitHub repository, which will be presented in more detail in section 3.2, contains several other patterns from different categories that have not been published in a scientific paper yet.

3.2 Published Pattern Catalogs

A commonality of all the published pattern catalogs featured in this section is the lack of a detailed description. Some provide an intention or the problem and solution domain of a pattern, but a throughout explanation of benefits and drawbacks is missing in all of them. While there are several other pattern catalogs out there, these are the most prominent or extensive ones:

A library with the intention to help writing secure smart contracts has been published by OpenZeppelin [50]. Their work features a multitude of tested and reviewed contracts from different categories that can be reused. Their repository is one of the most widespread Solidity resources and is still actively maintained until the time of writing. Minimal information about the functionality is given via inline comments.

GitHub user cjgdev published a library of eight smart contract patterns [51]. Each of the patterns is written in Solidity and accompanied by its intent. The patterns are distributed in four groups: Lifetime, Maintenance, Ownership and Security.

Some of the patterns published by cjgdev found its way into the catalog of Simoes [52], which is a compilation of eleven patterns from different sources, including from Simoes himself. Some of the patterns have undergone slight modifications before being put together to the referenced catalog, which only contains patterns. No categories or further explanations are provided.

The last repository to be mentioned is from Wöhrer, the author of one of the papers featured in the previous section. His repository is a compilation of 18 patterns from 5 different categories [53]: Action and Control, Authorization, Lifecycle, Maintenance and Security. The patterns from the security category are the ones featured in the aforementioned paper. A problem and solution is provided with each pattern.

Table 3.1: An overview of the work related to this thesis.

Author(s)	Description
Scientific Contributions	
Delmolino et al. [16]	Experiences in teaching smart contract programming to undergraduate students.
Bartoletti and Pompianu [46]	Identification of design patterns and quantification of smart contracts implementing them.
Zhang et al. [47]	Usage of familiar design patterns to resolve design specific challenges in a health care application.
Mavridou and Laszka [48]	Presentation of a tool for automatically generating smart contracts as finite state machines.
Wöhrer and Zdun [49]	Presentation of six Solidity security patterns.
Published Pattern Catalogs	
OpenZeppelin [50]	Several well tested and audited contracts that are kept up to date.
cjgdev [51]	Eight Solidity patterns in four categories with their intent.
Simoes [52]	Compilation of eleven Solidity patterns without description.
Wöhrer [53]	18 Solidity patterns in five categories, including their problem and solution domain.

4 Pattern Catalog

This chapter presents 14 selected patterns grouped into their corresponding categories. The featured patterns are not meant to provide a tailored solution to the underlying problems, but are supposed to be an inspiration and source of information, in order to implement a fitting solution for each respective case. The following catalog should not be misunderstood as a tutorial for Solidity. A basic understanding of Solidity syntax and smart contract programming is assumed to be present[1]. The code samples provided with the patterns were created by using Solidity in the version `0.4.21`.

To give an overview over the catalog, the following list contains each pattern together with its page number in parentheses and its intent:

- **Guard Check (28)** Ensure that the behavior of a smart contract and its input parameters are as expected.
- **State Machine (32)** Enable a contract to go through different stages with different corresponding functionality exposed.
- **Oracle (36)** Gain access to data stored outside of the blockchain.
- **Randomness (41)** Generate a random number of a predefined interval in the deterministic environment of a blockchain.
- **Access Restriction (48)** Restrict the access to contract functionality according to suitable criteria.
- **Checks Effects Interactions (51)** Reduce the attack surface for malicious contracts trying to hijack control flow after an external call.
- **Secure Ether Transfer (55)** Secure transfer of ether from a contract to another address.
- **Pull over Push (59)** Shift the risk associated with transferring ether to the user.
- **Emergency Stop (63)** Add an option to disable critical contract functionality in case of an emergency.
- **Proxy Delegate (68)** Introduce the possibility to upgrade smart contracts without breaking any dependencies.
- **Eternal Storage (72)** Keep contract storage after a smart contract upgrade.

[1]Helpful tutorials to start with Solidity can be found in [54, 55].

- **String Equality Comparison (77)** Check for the equality of two provided strings in a way that minimizes average gas consumption for a large number of different inputs.
- **Tight Variable Packing (81)** Optimize gas consumption when storing or loading statically-sized variables.
- **Memory Array Building (85)** Aggregate and retrieve data from contract storage in a gas efficient way.

4.1 Behavioral Patterns

When using a blockchain as a back end for one's service, one has to be aware of several quirks and limitations that this new technology brings with it. The immutability of contracts and transactions, the deterministic consensus protocol and the public accessibility of stored information are just some of them. Developers will have to get used to new paradigms and change their way of thinking to overcome certain limitations. To overcome some of these problems, we are introducing four patterns in this category. Each of them is supposed to be a little helper, aiding in the implementation of certain behavior that is not self-evident to implement in the context of a blockchain. They can be of help administering a contract, establishing a connection to the outside world, or overcoming the limitations of the deterministic consensus algorithm.

4.1.1 Guard Check

Intent

Ensure that the behavior of a smart contract and its input parameters are as expected.

Motivation

Like in a regular legal contract, it is often the case in smart contracts that contract logic is only supposed to come into effect after certain requirements are met. For example a heritage should only be paid out to the heirs after the testator is deceased. While we have lawyers and notaries in the real world, on the blockchain, without regulators or mediators, we require some sort of guards or checks in order to assure that smart contract logic is functioning as specified.

The desired behavior of a smart contract would be to check for all required circumstances and only proceed if everything is as intended. In case of any shortcomings, the contract is expected to revert all changes that have been made to its state. To achieve this, Solidity is making use of the way the EVM handles errors: to retain atomicity[2] all changes are reverted and the whole transaction is without effect. Solidity is using exceptions to trigger these errors and revert the state [2]. There are several ways provided by

[2]Either everything in a transaction is executed, or nothing is.

Solidity to trigger such exceptions. This pattern describes their differences and gives an idea on how and when to use each of them.

Applicability

Use the Guard Check pattern when

- you want to validate user inputs.
- you want to check the contract state before executing logic.
- you want to check invariants in your code.
- you want to rule out conditions that should not be possible.

Participants & Collaborations

While the pattern can be used to validate data submitted by users as well as data returned from other contracts, the only participant is the implementing contract itself, as all behavior is performed internally.

Implementation

Prior to Solidity version 0.4.10 checks were commonly implemented with an if-clause that would throw an exception in case the requirement is not met: `if(testator != deceased)throw;`. Since version 0.4.13 the keyword `throw` is deprecated [56] and the use of one of this three other functions is recommended: `revert()`, `require()` and `assert()`. How and when to use which of them to trigger exceptions will be discussed in this section.

Before the Byzantium update[3], `require()` and `assert()` behaved identically. Since then the underlying opcodes actually differ. The two methods `require()` and `revert()` use `0xfd` (`REVERT`) while `assert()` uses `0xfe` (`INVALID`). The big difference between the two opcodes is gas return. While `REVERT` is refunding all of the gas that has not been consumed at the time the exception is thrown, `INVALID` uses up all gas included in the transaction [11]. This difference should be kept in mind and already gives an indication for which situations they are intended for.

The Solidity documentation [2] suggests that `require()` "should be used to ensure valid conditions, such as inputs, or contract state variables [..], or to validate return values from calls to external contracts" and `assert()` "should only be used to test for internal errors, and to check invariants". Both methods evaluate the parameters passed to it as a boolean and throw an exception if it evaluates to `false`. The `revert()` method throws in every case. It is therefore useful in complex situations, like if-else trees, where the evaluation of the condition cannot be conducted in one line of code and the use of `require()` would not be fitting.

[3]A hard fork on the Ethereum blockchain on the 16.10.2017 to bring the first part of the Metropolis update.

Generally `require()` should be used towards the beginning of a function for validation and should be used more often than the other two. The `assert()` method will be used at the end of a function and should only prevent severe errors. Under normal circumstances and bug free code the `assert()` statement should never evaluate to `true`. [57]

Since Solidity version 0.4.22 it is possible to append an error message to `require(bool condition, string message)` and `revert(string message)` [2].

Sample Code

This fictional sample contract is a donation distributor. Users send the address of a charity they want to support and a donation in the form of ether. In case the charity has no ether on their address, the whole amount is forwarded. If they do already own some ether, but less than the donor, half the amount of the donation is transferred while the other half stays at the contract for future distribution (this functionality is not implemented for the sake of brevity). In case the charity has more funds than the donor, no money should be donated. This sample contract showcases all three possibilities to implement the Check Guard pattern.

```
 1  contract GuardCheck {
 2
 3      function donate(address addr) payable public {
 4          require(addr != address(0));
 5          require(msg.value != 0);
 6          uint balanceBeforeTransfer = this.balance;
 7          uint transferAmount;
 8
 9          if (addr.balance == 0) {
10              transferAmount = msg.value;
11          } else if (addr.balance < msg.sender.balance) {
12              transferAmount = msg.value / 2;
13          } else {
14              revert();
15          }
16
17          addr.transfer(transferAmount);
18          assert(this.balance == balanceBeforeTransfer - transferAmount);
19      }
20  }
```

Listing 4.1: Guard Check pattern sample code

The function `donate(address addr)` in line 3 takes an address as an input parameter and carries the `payable` modifier. This globally reserved keyword is used to allow a function to receive ether. Without it, any transaction calling this function, while carrying ether, would be rejected [2]. In line 4 the `require` statement makes sure that the address provided by the user is not zero as it would be the case, if the user would have forgotten to specify a charity. Line 5 then checks, if the user attached a donation to his transaction. If this is not the case, we can stop right there. The if/else construct starting in line 9 determines the amount to be sent to the charity, depending on the charities current balance. In case the charity has more ether than the donor, the `revert` in line 14 makes sure that no money is transferred and the function is reverted. In line 17 the donation is sent to the charity. The final `assert` statement in line 18 assures that the current balance after the donation is equal to the balance before the donation minus the donated amount. Under regular circumstances this should always evaluate to `true`. If this assertion would not hold true, the whole transaction including the donation transfer to the charity would be reverted.

Consequences

One positive consequence of applying the Guard Check pattern is increased readability. Compared to an if/throw construct the use of a `require` function makes it easier for the reader, who might not be a software engineer, to understand the intention of the operation. Additionally the new expressions look cleaner in general. Another benefit of having several options is that the individual methods allow for future functionality to be implemented, tailored to their purpose. As mentioned, the `revert()` and `require()` methods were equipped with an error message, while `assert()` could be used for evaluation purposes with techniques like static analysis and formal verification to identify conditions that break contract logic [57]. The availability of three different usable functions allows for a targeted application in various situations and therefore provides flexibility to the developers.

The wide range of possible methods can be confusing for users without any experience with this pattern, because the names suggest similar behavior, but do not offer any explanation on where they differ. Using the wrong statement can lead to undesired behavior, for example users loosing all their provided gas in case they make a typo in the function arguments.

Overall the Guard Check provides a reliable way to handle errors and guard against undesired behavior. It is therefore an essential ingredient in the Access Restriction pattern, as described in subsection 4.2.1.

Known Uses

The application of this pattern can be observed in nearly every published contract. A nice example is the contract of HODLit[4], a token with the intention to give an incentive to hold ether, because it features all three methods. The `require` expression is used for checks at the beginning of methods and `assert` is used to make sure that arithmetic operations cannot over- or underflow. The `revert` method is called in the fallback function in line 269 to avoid the possibility to send ether to the contract without calling one of the functions specified for that purpose.

A negative example can be observed in this simple casino contract[5]. The developer used `assert` for every check in the whole contract. This would lead to the loss of all

[4]HODLit: `https://etherscan.io/address/0x24021d38DB53A938446eCB0a31B1267764d9d63D`
[5]Casino: `https://github.com/merlox/casino-ethereum/blob/master/contracts/Casino.sol`

provided gas, if one of the checks fails. In case a user wants to make sure his transaction does not run out of gas and therefore provides a very high gas limit, this could result in the loss of a significant amount of money.

4.1.2 State Machine

Intent

Enable a contract to go through different stages with different corresponding functionality exposed.

Motivation

Consider a contract that has to transition from an initial state, over several intermediate states, to its final state over his lifetime. At each of the states the contract has to behave in a different way and provide different functionality to its users. The described behavior can be observed in a multitude of use cases: auctions, gambling, crowdfunding and many more. Even the Solidity documentation acknowledges its conventionality by listing it as one of the common patterns [2]. There are different ways one state can transition into another. Sometimes a state ends with the end of a function, another time the state is supposed to transition after a specified amount of time. The different possibilities will be described in greater detail in the Implementation section.

A pattern with the functionality described above has already been formulated by Gamma et al. [43], but its implementation on a blockchain is especially interesting. This is because a blockchain itself is a state transition system [9], where an initial state in combination with a transaction has a new state as an output. To avoid confusion between the states of a blockchain (i.e. the condition before and after a transaction) and the states, a smart contract is going through, we will call the explicitly defined states of contracts *stages* from here on.

Applicability

Use the State Machine pattern when

- a smart contract has to transition several stages during its life cycle.
- functions of a smart contract should only be accessible during certain stages.
- stage transitions should be clearly defined and not preventable for all participants.

Participants & Collaborations

There are two participants in the State Machine pattern. The first participant is the implementing contract that will transition through the predefined stages and guarantees that only the intended functions in the respective stage can be called. The other participant is the contract owner or interacting users, which are able to initiate a stage transition, either directly or indirectly through timed transitions.

Implementation

The implementation of the State Machine pattern covers three main components: **representation of the stages**, **interaction control for the functions** and **stage transitions**.

To model the different stages in Solidity we can make use of enums. Enums are user-defined data types [2]. After one enum containing all possible stages is declared, an instance of that enum can be used to store the current stage and transition to the next one by assigning it a new stage. Since enums are explicitly convertible to and from all integer types, a transition to the next stage can be accomplished by adding the integer `1` to the stage instance.

The restriction of function access to certain stages can be achieved by using the Access Restriction pattern from subsection 4.2.1. A function modifier checks, if the contract stage is equal to the required stage before executing the called function. In case the function is called at an improper stage, the transaction is reverted using the Guard Check pattern.

There are several ways to transition from one stage to the next one, in order to accommodate for different situations and needs. One way is the transition in a function call. Either the function exists exclusively for the stage transition, or it executes business logic and the stage transition is a natural part of the process. For example in a roulette contract, the house could call a function to pay out all winnings, which ends with a stage transition from `GameEnded` to `WinnersPaid`. In cases like this, the stage change is either implemented directly by assigning a new stage to the state variable, by using a modifier initiating the transition at the end of the function, or by a helper function. The helper function would be an internal function that increments the stage by 1, every time it is called. Another option that does not rely on direct function calls are automatic timed transitions. The duration a stage is supposed to last, or rather a point of time in the future at which the stage transition should be executed, is stored in the contract. A modifier that is called with every involved function call, checks for the current timestamp and transitions to the next stage, in case that point in time is already reached. It is important to mention that the order of the modifiers in Solidity matters [2]. With this knowledge in mind, the modifier for the timed transition should be mentioned before the modifier that checks for the current stage, to make sure that the potential timed stage change is already considered in the stage check.

After the implementation is done, heavy testing is necessary to rule out any possibilities for unexpected stage changes by malicious entities that could try to benefit from unintended behavior or just break the contract.

Sample Code

This sample contract showcases the state machine for a blind auction and is inspired by the example code provided in the Solidity documentation [2]. It features stage transitions in functions, as well as timed transitions. As this is an extensive use case, only the code relevant for the state machine is presented. Any logic dealing with the auction, including the storage of bids, is omitted.

```
contract StateMachine {

    enum Stages {
        AcceptingBlindBids,
        RevealBids,
        WinnerDetermined,
        Finished
    }

    Stages public stage = Stages.AcceptingBlindBids;

    uint public creationTime = now;

    modifier atStage(Stages _stage) {
        require(stage == _stage);
        _;
    }

    modifier transitionAfter() {
        _;
        nextStage();
    }

    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindBids && now >= creationTime + 6 days) {
            nextStage();
        }
        if (stage == Stages.RevealBids && now >= creationTime + 10 days) {
            nextStage();
        }
        _;
    }

    function bid() public payable timedTransitions atStage(Stages.AcceptingBlindBids) {
        // Implement biding here
    }

    function reveal() public timedTransitions atStage(Stages.RevealBids) {
        // Implement reveal of bids here
    }

    function claimGoods() public timedTransitions atStage(Stages.WinnerDetermined)
         transitionAfter {
        // Implement handling of goods here
44      }
45
46      function cleanup() public atStage(Stages.Finished) {
47          // Implement cleanup of auction here
48      }
49
50      function nextStage() internal {
51          stage = Stages(uint(stage) + 1);
52      }
53  }
```

Listing 4.2: State Machine pattern sample code

In line 3 the enum `Stages`, which contains the four stages the auction will be going through, is defined. A state variable is initialized with the initial stage in line 10. The time of creation is stored in line 12 and will be important for the timed transitions. The function modifier defined in line 14 is checking the current stage versus the allowed stage of a function. The stage provided as a parameter is the stage that the contract has to be in, in order to be able to execute the function logic. If a function implements the modifier `transitionAfter()`, the internal method `nextStage()` is called at the end of the function and the contract transitions in the next stage. Timed transitions are handled by using the modifier specified in line 24. The individual if-clauses check, if the contract should already be in the next stage by comparing the current time with the time the transition is supposed to happen (e.g. `creationTime + 6 days`), while also taking the current stage into account.

The four public methods starting from line 34 are only callable in their respective stages, which is achieved by the `atStage()` modifier provided with the concrete stage. Transitions for the first two stages are timely. Notice that the `timedTransitions` modifier is included in the first three functions and not only in the first two. This is because the actual transition is happening when calling the function of the next stage. For example: calling the `bid()` function 8 days after contract creation will first transition to the next stage, because the `timedTransitions` modifier triggers. Afterwards, however, the `atStage()` modifier will detect that the stage is not matching anymore and will revert the whole transaction, including the stage transition. In this case the `timedTransitions` modifier is making sure that `bid()` cannot be called after 6 days have passed. The persisting stage transition is happening the first time the function of the next stage is called, in this case `reveal()`. The process is the same as before, only that this time the stage is recognized as correct and the transaction is not reverted. This complex behavior is why the order of the modifiers mentioned in the Implementation section is so important.

The transition from stage three to four is done with the `transitionAfter` modifier used in the function in line 42. After the execution of the function, the contract automatically goes into the next and final stage, which only allows the `cleanup()` method to be called.

Consequences

One consequence of applying the State Machine Pattern is the partitioning of contract behavior into the distinct stages. It allows functions to be only called at the intended times. Additionally, the pattern guides the contract through its different stages by providing several options for initiating stage transitions that can be used depending on the context.

Some consequences that should be kept in mind, emerge from the different options for stage transitioning. While timed transitions have the benefit of a clear policy for every participant, the usage of block numbers or timestamps is not entirely harmless. Miners have the potential ability to influence timestamps to a certain degree. It should therefore be avoided to use timed transitions for very time sensitive cases. To be completely safe, a contract should be robust against a timestamp deviating by up to 900 seconds from the actual time [58]. But also the manual transition of stages by the contract owner is prone to manipulation, as the owner can simply decide to change stages in his favor or abandon the contract and freeze all invested funds.

Known Uses

Several contracts applying this pattern in some form or another could be observed. One example is the betting contract from Ethorse[6], which allows bets on the price development of cryptocurrencies. In this example, the stages are stored in a struct in the form of booleans, with the current stage being set to `true`. Transitions are made in a timely fashion via timestamps. A different implementation that relies mostly on manual transitions is the auction contract of Pocketinns[7], a community driven marketplace ecosystem. In this contract the owner has the ability to change stages at his own will. Even though it is described as an emergency measure in the contract, it opens the door for severe manipulation and any transaction with such a contract should be done with care.

4.1.3 Oracle

Intent

Gain access to data stored outside of the blockchain.

Motivation

Every computation on the Ethereum blockchain has to be validated by every participating node in the network. It would not be practical to allow for any kind of external network

[6]Ethorse contract: `https://github.com/ethorse/ethorse-core/blob/master/contracts/Betting.sol`

[7]Pocketinns contract: `https://github.com/pocketinns/PocketinnsContracts/blob/master/DutchAuction.sol`

requests from a contract, since every node would have to make that request on their own to verify its result. Not only would this lead to excessive network usage, which smaller sites could possibly not handle, also a change in the requested information would break the consensus algorithm. Contracts on the blockchain itself are therefore not able to communicate with the outside world, meaning they cannot pull information from sources like the internet. But for many contracts, information about external events is necessary to fulfill their core purpose, especially since the industry is looking for more complex use cases. There are already contracts that rely on information about the result of a sport event [59], the price of a currency [60] or the status of a flight [61]. One of the first solutions to overcome this limitation was Orisi [62], a service published in 2014, with the aim of being an intermediary between the Bitcoin blockchain and the outside world. Since then several services for different blockchains with similar concepts have emerged under the term oracle. The oracle acts as an agent living on the blockchain and providing information in the form of responses to queries.

An important point when handling data in the context of a blockchain is the notion of trust. When relying on externally introduced information for a centralized source, it is necessary to find a way to build up trust for that information as well.

Applicability

Use the Oracle pattern when

- you rely on information that cannot be provided from within the blockchain.
- you trust the provider of the necessary information.

Participants & Collaborations

The oracle pattern consists of three entities: the contract requesting information, the oracle and the data source [63]. The oracle itself is usually consisting of a smart contract and an off-chain[8] server. The process begins with a contract requesting information, which he cannot retrieve from within the blockchain. Therefore, a transaction is sent to the oracle contract, which lives on the blockchain as well. This transaction contains a request that the contract wishes to be fulfilled. Optional parameters can be the desired data source, or a certain time in the future, at which the answer should be delivered.

Afterwards the oracle server is notified by the incoming request, for example by listening to events [2], and forwards the request to the specified data source. Because the oracle server, as well as the data source are off-chain, this communication is not conducted via blockchain transactions, but through other forms of digital communication.

When the request reaches the data source it is processed and the reply is sent back to the oracle server. From there on the oracle contract is invoked by the server, to either send the result back to the requesting contact or wait until the time stated in the request.

[8]Not implemented on the blockchain (e.g. a website with a REST API)

The initial contract receives the data via a function call in which the contract can then execute any logic on the data. A graphical representation of this composition is shown in Figure 4.1.

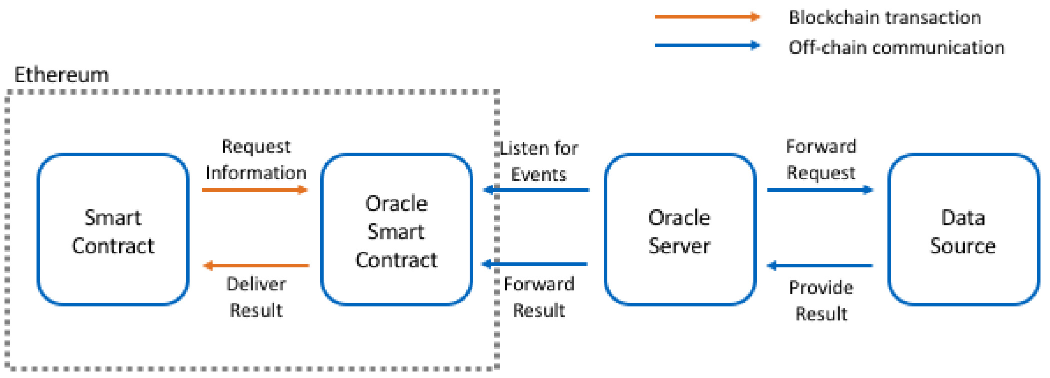

Figure 4.1: Participants and communication of a smart contract using the Oracle pattern.

Implementation

In this section we will only focus on the implementation of the pattern in the requesting contract. The implementation of the oracle itself is done mainly off-chain and is therefore not covered here. There are several resources on the internet that cover how to implement an oracle, customized for the needs of one's smart contract and business model. However, people thinking about interacting with a contract might be turned away by seeing that self-provided data is being used for the execution of contract logic. They would have to trust the contract creator, who in this scenario is the same entity as the oracle operator, to not manipulate the data on the way from the data-source to the contract. This reintroduces the need for trust, which we try to get rid of by using a blockchain.

The more commonly used alternative to this is using an independent service as an oracle. At this point, the market leader in this domain is the British company Oraclize [64]. Other oracle services are for example Town Crier[9], working with trusted hardware, or Reality Keys[10].

Whether the oracle is self implemented or an external service is used, it is necessary for the requesting contract to implement at least two methods:

1. The first method assembles a query to let the oracle know which data is requested and sends it in a transaction to the oracle contract. Depending on the implementation of the oracle additional parameters can be added to the request. It is common, that the oracle returns an ID that can be stored for future reference.

[9]Town Crier: `http://www.town-crier.org/`
[10]Reality Keys: `https://www.realitykeys.com/`

2. The second method is a so called callback function. This is the function called by the oracle contract to deliver the result of the query. The callback function either stores the result of the query or triggers any internal logic. The incoming calls can be associated by the ID returned in the first method. It makes sense to include a check in the callback function to make sure that only the oracle is able to call it. Otherwise a malicious entity could provide wrong information to do harm or benefit from it.

Sample Code

As the service Oraclize is used in almost every case an oracle is needed, the following sample will showcase the code needed to interact with the Oraclize oracle to receive the current EUR to USD exchange rate. Other oracles are integrated in a similar fashion. Information about the accurate syntax needed, can be found in the respective documentation.

```
 1  import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";
 2
 3  contract OracleExample is usingOraclize {
 4
 5     string public EURUSD;
 6
 7     function updatePrice() public payable {
 8         if (oraclize_getPrice("URL") > this.balance) {
 9             //Handle out of funds error
10         } else {
11             oraclize_query("URL", "json(http://api.fixer.io/latest?symbols=USD).rates.USD"
                   );
12         }
13     }
14
15     function __callback(bytes32 myid, string result) public {
16         require(msg.sender != oraclize_cbAddress());
17         EURUSD = result;
18     }
19  }
```

Listing 4.3: Oracle pattern sample code

In line 2 the API of Oraclize is imported from GitHub. In case the compiler is not supporting the direct import from sources like GitHub, it is necessary to replace the import statement with a local import of the API. The API is needed to give access to addresses and the functions needed to interact with the oracle. In line 4 it is specified that the contract inherits from the API by using the keyword `is`. The function `updatePrice()` is sending out the query to the oracle. The `payable` modifier is used, which allows the transaction to have a value (some amount of Ether) attached. This is necessary because the usage of the Oraclize service is not for free. The applying rates can be found in the Oraclize documentation [64]. Line 9 asserts that the contract has sufficient funds to pay for the service. If this is not the case the user should be notified, for example by triggering an event. If the balance is sufficient, the query is sent to the Oraclize contract in line 12. The first parameter tells the oracle that we want to query a URL while the second parameter contains the URL of the API and the part of the response JSON object we are interested in, namely the USD value. Any API on the internet can be accessed by this way.

The `__callback(bytes32 myid, string result)` function is used by the oracle to send the result to the contract. The first paramter `myid` could have been saved in the first function and now be used to link the result to a previous request. Line 17 makes sure the calling entity is indeed the oracle. The result is then saved to storage in line 18.

Consequences

The most important consequences of applying the oracle pattern is gaining access to data otherwise not being available on the blockchain, and therefore allowing business models and use cases with additional functionality. Besides providing arbitrary data from the web, oracles can be used to automatically trigger a function at a specified time in the future, by providing a time delay parameter in the query. This can solve the often encountered problem of how to schedule function calls on a blockchain [65]. It is also often used for generating random numbers, a difficult task, as described in subsection 4.1.4. From a developer standpoint it is fairly easy to implement the oracle pattern, especially when using one of the already existing services. Another benefit of using an existing solution is the fact that these solutions are heavily audited, reducing the risk of errors.

A negative consequence of the usage of oracles is the introduction of a single point of failure. The contract creator as well as the users interacting with the contract rely heavily on the information provided by the oracle. Oracles or their data sources have reported wrong data in the past [66] and it is likely that there will be errors in the future again. Not only errors but even small changes in the format of the provided data can break a smart contract like it happened in a case published on reddit [67]. A data source changed the formatting of its outputs for a boxing fight from lowercase to uppercase characters, which the immutable smart contract could not handle anymore. Another negative consequence is the trust that has to be put into both, oracles and data sources. In an environment that strives towards decentralization, relying on a single external entity seems contradictory. This issue could potentially be mitigated by forwarding the request to a couple of independent oracles. The results would then be compared and evaluated. A possible strategy could be using M independent oracles and only accepting a result reported by at least N (with N < M) of the M agents. A drawback of this approach is the cost that increases with every additional oracle. Also the time it takes to come to a conclusion is increasing in most of the cases, since one will have to wait for at least N responses. Another method of mitigating trust from the oracles is beeing used at Oraclize: TLSNotary proofs [68]. Using TLSNotary, Oraclize can prove that they visited the specified website at a certain time and indeed recieved the provided result. While this would not prevent Oraclize from querying random numbers until they get the desired result, it is trustworthy in case the requested data does not fluctuate over small periods of time.

Further trust issues could be resolved in the future with the adoption of decentralized oracles, on which a lot of work is done during the time of writing [69, 70].

Known Uses

Usage of the oracle pattern can be observed in a variety of contracts on the blockchain. An example using the service Oraclize is the contract of Etherisc[11] where the oracle is used to get access to flight delay data. The contract then pays out insurance to their users in case of a delayed flight.

Another oracle implementation can be observed in Ethersquares[12], a sports betting contract. In this case the owner of the contract acts as an oracle himself. Users are able to verify, if the results provided by the owner are correct, with the help of a voting mechanism.

The research conducted in section 6.3 revealed that at the time of writing, a little less than 1% of newly created smart contracts make use of the service Oraclize as an oracle. A relatively high number, considering there are other oracle providers on the market, like the ones mentioned above, which further add to that number.

4.1.4 Randomness

Intent

Generate a random number of a predefined interval in the deterministic environment of a blockchain.

Motivation

Randomness in computer systems and especially in Ethereum is notoriously difficult to achieve. While it is hard or even impossible[13] to generate a truly random number via software [11], the need for randomness in Ethereum is high. This stems from the fact that a high percentage of smart contracts on the Ethereum blockchain can be classified as games [46], which often rely on some kind of randomness to determine a winner. The problem with randomness in Ethereum is that Ethereum is a deterministic Touring machine, with no inherent randomness involved. A majority of miners have to obtain the same result when evaluating a transaction to reach consensus. Consensus is one of the pillars of blockchain technology and randomness would imply that mutual agreement

[11]Etherisc: `https://github.com/etherisc/flightDelay/blob/master/contracts/FlightDelayPayout.sol`

[12]Ethersquares: `https://github.com/ethersquares/ethersquares-contracts/blob/master/contracts/OwnedScoreOracle.sol`

[13]Some solutions therefore only provide pseudorandomness, a process that appears to be random, but is not.

between all nodes is impossible. Another problem is the public nature of a blockchain. The internal state of a contract, as well as the entire history of a blockchain, is visible to the public [71]. Therefore, it is difficult to find a secure source of entropy. One of the first sources of randomness in Ethereum that came to mind were block timestamps [72]. The problem with block timestamps is, that they can be influenced by the miner, as long as the timestamp is not older than its parent block [58]. Most of the time the timestamps will be close to correct, but if a miner has an incentive to benefit from wrong timestamps, he could use his mining power, in order to mine his blocks with incorrect timestamps to manipulate the outcome of the random function to his favor.

Several workarounds have been developed that overcome this limitations in one way or the other. They can be differentiated into the following groups [71], each with their respective benefits and downsides:

- **Block hash PRNG**[14] - the hash of a block as source of randomness
- **Oracle RNG**[15] - randomness provided by an oracle, see subsection 4.1.3
- **Collaborative PRNG** - collaborative generation of a random number within the blockchain

Because the use of an oracle has already been discussed in the respective pattern and the most renown example of collaborative PRNG, Randao, is not being actively developed anymore [73], we will focus on the generation of pseudorandom numbers with the help of block hashes in this chapter. Considerations between the use of oracle RNG versus block hash PRNG will be discussed in the Consequences section.

Applicability

Use the Randomness pattern when

- you want to generate a random number that is not predictable by the users.
- you do not want to use any external services for randomness.
- you have a trusted entity that is able to reliably provide seeds for the generation of randomness.

Participants & Collaborations

The participating entities in this pattern are the calling contract, a trusted entity and a miner, mining the block of which we are using the block hash as source of entropy. The contract makes use of the globally available variable of the hash of a block [2] and uses it together with a seed, provided by the trusted entity, to internally compute a number that should be unknown to anyone until the block is mined.

[14]Pseudorandom number generator
[15]Random number generator

Implementation

The simplest implementation of this pattern would be just using the most recent block hash:

```
function randomNumber() internal view returns (uint) {
    return uint(blockhash(block.number - 1));
}
```

Listing 4.4: Randomness pattern simple implementation

Implemented like this there are two problems, making this solution impractical:

1. A miner could withhold a found block, if the random number derived from the block hash would be to his disadvantage. By withholding the block, the miner would of course lose out on the block reward. This problem is therefore only relevant in cases the monetary value relying on the random number is at least comparatively high as the current block reward.

2. The more more concerning problem is that since `block.number` is a variable available on the blockchain, it can be used as an input parameter by any user. In case of a gambling contract, a user could use `uint(blockhash(block.number - 1)` as the input for his bet and always win the game.

To get rid of the possibility of interference by miners and prediction of random numbers, Bonneau et al. proposed a solution applied on the Bitcoin blockchain [74]: a trusted party provides a seed, which will be hashed together with a future block hash, to make it impossible for the miner to predict the outcome of his block hash on the random number. We are using this idea in this pattern to avoid interference by malicious miners.

The trusted party can be chosen by the contract creator and is stored in the contract. In the beginning, users can make their interaction with the contract (like placing bets) in the first stage. With the submission of the sealed seed by the trusted party, bets are closed and the current block number + 1 is stored, which will come in handy later. The seed can be sealed by hashing it together[16] with the address of the trusted party. This allows for easy validation in the next step.

After the seed has been stored, the trusted party has to wait for at least one block until it can reveal the seed. Of course, it has to be validated that the committed hash was the result of a hash of the now provided seed, by comparing the sealed seed with the hash of the actual seed and the address of the trusted party. If this is the case, the seed is accepted and can be hashed together with the stored block number to generate a pseudorandom number. We use the block number stored in the previous step, because using the current block number would allow for interference by withholding from the miner again, as the seed is sent in plaintext. With the incrementation the block number before storing it, we are making sure a future block hash is used as source of entropy, making it impossible for the trusted party to predict it.

[16]Keccak-256, the standard hash function in Ethereum, is used for hashing.

In case the random number is supposed to be of a special interval, the modulo function can be utilized. Depending on the desired length, only the last part of the obtained hash is used.

Sample Code

The provided sample showcases the implementation of a pseudorandom number generator with the use of a trusted entity in the context of a betting contract. Any logic regarding the betting process is omitted for the sake of clarity.

```
1  contract Randomness {
2  
3      bytes32 sealedSeed;
4      bool seedSet = false;
5      bool betsClosed = false;
6      uint storedBlockNumber;
7      address trustedParty = 0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF;
8  
9      function setSealedSeed(bytes32 _sealedSeed) public {
10         require(!seedSet);
11         require (msg.sender == trustedParty);
12         betsClosed = true;
13         sealedSeed = _sealedSeed;
14         storedBlockNumber = block.number + 1;
15         seedSet = true;
16     }
17 
18     function bet() public {
19         require(!betsClosed);
20         // Make bets here
21     }
22 
23     function reveal(bytes32 _seed) public {
24         require(seedSet);
25         require(betsClosed);
26         require(storedBlockNumber < block.number);
27         require(keccak256(msg.sender, _seed) == sealedSeed);
28         uint random = uint(keccak256(_seed, blockhash(storedBlockNumber)));
29         // Insert logic for usage of random number here;
30         seedSet = false;
31         betsClosed = false;
32     }
33 }
```

Listing 4.5: Randomness pattern sample code

The trusted party is hard-coded into the contract in line 7. It would be an option for the owner to permit change of the trusted party with the help of a setter function protected against unauthorized access by the Access Restriction pattern from subsection 4.2.1. Users can make their bets by calling the function `bet()`. The hashed seed can be set by the trusted party, and only the trusted party (line 11), by calling `setSealedSeed(bytes32 _sealedSeed)`. With the function execution, the sealed seed as well as the incremented current block number is stored and the `seedSet` boolean is set to true to avoid the seed being overwritten by a second function call. Additionally, bets are closed to avoid that the trusted party or the miner can push their bets after learning about the seed or the block hash used to generate the random number.

After at least one block has passed, subsequently providing the sealed seed, the trusted entity can reveal the seed by calling `reveal(bytes32 _seed)` in line 23. The lines 24-27 implement the Guard Check pattern and assure that the seed can only be revealed after the sealed seed was set (line 24), the relying action has been performed (line 25) and the block we are referencing has already been mined (line 26). An access restriction for the trusted party could be implemented but is not mandatory, as the trusted party should be the only entity that can provide a seed which matches the sealed seed. This is verified in line 27, where it is checked , if the seed provided by the trusted party was indeed the same as the one committed in the step before. The actual random number is generated in line 28 by hashing the seed together with the hash of the block at the previously stored number. Next steps could be the formatting of the number into the desired interval and execution of any logic using the random number, like the payout of the winners.

Consequences

The consequences of the Randomness Pattern can be evaluated after the following criteria inspired by [71]:

- **Randomness** - how good is the achieved randomness? Is it pseudo or true randomness?
- **Security** - how secure is the used method to generate randomness?
- **Cost** - how high are the costs associated with generating randomness?
- **Delay** - how big is the time delay between request and reception of the random number?

The **randomness** generated by the proposed method is pseudorandom. The block hash as well as the seed are provided in a deterministic way and if both input parameters were known, the result could be predicted. However, due to the combination of block hash and seed from two different sources, and both sources having to commit their inputs before learning of the other, it is practically impossible to influence the random number for one's benefit.

Once a random number is obtained, we can assume that it is **secure**. The only form of insecurity is introduced by the trusted party. The name trusted party does not mean that we have to trust the party blindly. On the contrary, the measures taken make it impossible to manipulate the random number, even for the trusted party. We only have to trust it to reveal the provided seed. As the seed is sealed in a cryptographically secure way, there is currently no possibility to obtain the seed without the trusted party. Additionally, the Ethereum blockchain only allows access to the 256 most recent blocks [2], meaning that the trusted entity has to reveal the seed before the stored block number is not retrievable anymore. A revert mechanism for this case, which lets the users retrieve their funds, should be implemented. In summary, this means that the only way of cheating, with this pattern implemented, would be for the trusted party to withhold the revealed seed, or if the trusted party could influence the block creation of the block, which we are using the hash of (either by mining itself or colluding with miners). Nevertheless, this is an improvement over the previous solution, as there is now only one single potential threat, compared to several miners from before.

The **costs** of this method are relatively low, as no external service has to be payed. The gas requirements using a trusted entity are higher, compared to the simple case, as more transactions and storage is needed.

Due to the commitment to a seed and the use of a future block hash, the generation of the random number comes with a little **delay**. In the fastest case a result can be expected after two blocks.

When we compare these consequences with the ones of the Oracle pattern, we can work out their differences. The randomness provided by the Oracle can be true randomness, as we can query numbers from services providing true random numbers. While we only have to trust one party in our example, two parties have to be trusted when interacting with oracles: the data provider as well as the oracle service. Another difference is that the oracle service has to be paid for each request. The delay experienced with the oracle solution is comparable to the one proposed above.

It can be concluded that in simple contracts with no financial impact, a simple implication of block hash randomness without a seed is sufficient. For use cases with higher stakes an oracle service or the showcased solution with a seed can be used, depending on the trust one is willing to put into other parties.

Known Uses

Randomness is often used in contracts with a gaming or gambling context. Implementation of randomness via a future block hash and a seed can be observed in the Cryptogs contract[17], a DApp that provides a version of the game of pogs on the Ethereum blockchain. A commit/reveal scheme is used to avoid the use of an oracle. However, they claim that the added security, compared to a simpler implementation without a commit/reveal mechanic is not worth it for their use case [75]. The additional time and costs related to the extra transactions are not in relation to the monetary value they are handling.

Even though trust is an issue, a lot of contracts seem to be using the services of oracles to access random numbers. All of the observed ones were using Oraclize as their service. The actual source of randomness that Oraclize is getting its numbers

[17]Cryptogs contract: `https://etherscan.io/address/0xeFabE332D31c3982B76F8630a306C960169bD5b3#code`

from is more heterogeneous. An example for a contract using Oraclize in combination with random.org is vDice[18], which claims to be the most popular Ether betting game with over 70.000 bets played. Another contract relying on the services of Oraclize is Pray4Prey[19]. In contrast to vDice, random numbers are generated at Wolfram | Alpha.

The general impression is that simpler contracts tend to rely on block hashes and therefore avoid external communications, while more sophisticated contracts and the ones dealing with larger stakes seem to be more likely to use the services of oracles.

4.2 Security Patterns

Smart contracts are a worthwhile target for attackers because of the following three reasons [45]:

- **Transactions are final:** Once an attack has been carried out successfully, there is no way to revert the attack without a hard fork[20] of the whole blockchain.
- **Difficult to prosecute:** Due to limited regulation and pseudonymity on the blockchain, attacks on smart contracts stay mostly without legal consequences.
- **Availability of byte/source code:** Since data stored on a blockchain is publicly accessible, it is possible to study contract byte code and in some cases even source code[21] for vulnerabilities.

In addition to that, Bartoletti and Pompianu [46] found out that a majority of smart contracts published on the Ethereum blockchain can be classified as financial contracts, meaning they are handling financial assets and are therefore likely to have a direct financial consequence once security is breached.

As shown in subsection 2.4.1, several attacks have already occurred to smart contracts on Ethereum and vulnerabilities are actively searched for by attackers. The multitude of different possible attacks that have to be taken into account when developing a smart contract, gives a good idea of how difficult it is to write solid code with Solidity.

The patterns presented in this chapter can help smart contract developers write secure code, by helping to understand built-in mechanics and functions, as well as providing proven and tested code frameworks, which can be adapted to suit the corresponding use case. Wöhrer and Zdun [49] published a similar approach, by presenting a collection of security patterns. This work differs in the selection of the patterns, even though some of the presented patterns occur in both researches, and the format in which the single patterns are presented.

[18]vDice contract: `https://etherscan.io/address/0x7DA90089A73edD14c75B0C827cb54f4248D47eCc#code`

[19]Pray4Prey contract: `https://etherscan.io/address/0xe648ae88a6d9b3373e115e3414be91b7cf12de4c#code`

[20]A hard fork is a permanent divergence from the previous version of the blockchain.

[21]Source code can be made available by contract owners on explorers like etherscan.io.

4.2.1 Access Restriction

Intent

Restrict the access to contract functionality according to suitable criteria.

Motivation

Due to the public nature of the blockchain, it is not possible to guarantee complete privacy for one's contract. One will not be able to prevent someone from reading the state of a contract from the blockchain[22], since everything is publicly visible for everyone. What can be done, is restricting read access to the state of one's contract by other contracts [2]. This is achieved by declaring the state variables as `private`. Also, functions can be declared `private`, nonetheless, doing so would prevent everyone outside the contract scope from calling it under any circumstances. Simply declaring them `public` , however, would open up access to every participant in the network. Most of the times it is desired to allow access to functionality in case certain rules are met. Often the access is supposed to be restricted to a defined set of entities, like the administrators of a contract. Other restrictions should only allow access at a special point in time or if the accessing entity is willing to pay a price for access. All these restrictions, and many more [76], can be realized by the implementation of the Access Restriction pattern and therefore grant security against unauthorized access to smart contract functionality.

Applicability

Use the Access Restriction pattern when

- your contract functions should only be callable under certain circumstances.
- you want to apply similar restrictions to several functions.
- you want to increase security of your smart contract against unauthorized access.

Participants & Collaborations

The participants in this pattern are the calling entity of the function to be restricted, and the contract the function belongs to. The calling entity can either be a user or another contract and invokes the function by sending a transaction to the respective contract address. The actors involved in the called contract are the restricted function as well as an additional component that is responsible for the actual access control.

[22]Encryption is the only way to handle sensitive data on the Ethereum blockchain.

Implementation

For the implementation of the Access Restriction pattern we are using the Guard Check pattern from subsection 4.1.1. The functionality provided by the Guard Check pattern allows us to check for the required circumstances once a function is called, and throws an exception, in case they are not met. One could argue that the checks could be placed at the beginning of the corresponding function as well. However, since these checks are often reused for more than one function, it is good practice to use modifiers[23], which can then be applied to the functions needing them. Modifiers can take arguments from the input parameters of the respective function, be provided with their own arguments, or have the condition hard-coded in their body, which limits reusability.

The structure of these modifiers usually follows an identical pattern: In the beginning the required condition is checked. Afterwards the execution jumps back to the initial function. This behavior is indicated by an underscore (`_;`) in the code of the modifier. If there is extra behavior that should be executed after the function, it is possible to insert additional code following the underscore in the modifier. This pattern, in particular the condition check in the beginning of the modifier, can be adapted in many ways to provide a variety of different access restrictions.

Sample Code

The following code features three different kinds of access restrictions in an example contract owned by an owner and is influenced by the example from the Solidity documentation [2]. Ownership can either be transferred by the current owner himself or be bought by anybody for 1 ether, after a month has passed since the last change in ownership.

```
pragma solidity ^0.4.21;

contract AccessRestriction {

    address public owner = msg.sender;
    uint public lastOwnerChange = now;

    modifier onlyBy(address _account) {
        require(msg.sender == _account);
        _;
    }

    modifier onlyAfter(uint _time) {
        require(now >= _time);
        _;
    }

    modifier costs(uint _amount) {
```

[23]Modifiers let one wrap additional functionality to a method, similar to the decorator patern in object oriented programming languages [2, 43].

```
19         require(msg.value >= _amount);
20         _;
21         if (msg.value > _amount) {
22             msg.sender.transfer(msg.value - _amount);
23         }
24     }
25
26     function changeOwner(address _newOwner) public onlyBy(owner) {
27         owner = _newOwner;
28     }
29
30     function buyContract() public payable onlyAfter(lastOwnerChange + 4 weeks) costs(1
           ether) {
31         owner = msg.sender;
32         lastOwnerChange = now;
33     }
34 }
```

Listing 4.6: Access Restriction pattern sample code

In line 5-6 the state variables `owner` and `lastOwnerChange` are initialized at contract creation time with the creator of the contract and the current timestamp. The first modifier `onlyBy(address _account)` in line 8 is attached to the `changeOwner(..)` function from line 26 and makes sure that the initiator of the function (`msg.sender`) is equal to the variable provided in the modifier call, which in this case is `owner`. The usage of this modifier leads to an exception being thrown, every time the guarded function is called by anybody besides the current owner.

The second modifier `onlyAfter(uint _time)` works in the same way, with the difference that it throws if the function it is attached to is called before the specified time. It is used in line 30 and is provided with the time of the last change of ownership plus four weeks[24]. Therefore, guaranteeing that the function call can only be successful after at least four months have passed since the last change.

The third and last modifier `costs(uint _amount)` in line 18 takes an amount of currency as an input and makes sure that the value provided with the calling transaction is at least as high as the specified amount, before jumping into the execution of the guarded function. This modifier differs from the other two, as it has code implemented after the function execution, which is triggered by the underscore in line 20. The additional if-clause starting in line 21, checks if more money than necessary was provided in the transaction and transfers the surplus amount back to the sender. This is a good example for the various possibilities modifiers can provide. Combined with the previous modifier the contract can only be bought by a new owner, after four weeks have passed since the last change in ownership, and if the transaction contains at least one ether. The `payable` modifier of the `buyContract()` function in line 30 is needed in order to be able to receive money with a transaction.

It should be noted that in this simple example, we could have omitted the modifiers and implement their code directly in the respective function bodies, without loosing any

[24]Four weeks are added instead of one month because Solidity does not support months as a unit of time.

functionality and even reducing complexity. The benefit of outsourcing the functionality into modifiers becomes apparent, as soon as two or more functions share the same, or similar requirements (like in the State Machine pattern from subsection 4.1.2), as the modifier allows for easy reusability.

Consequences

Several consequences have to be taken into account when applying the Access Restriction pattern. One controversial point is the readability of code. On the one hand, modifiers can make the code easier to understand, because the restriction criterion is clearly recognizable in the function header, especially if the modifiers are given meaningful names like in the provided sample code. On the other hand, execution flow is jumping from one line in the code to a completely different one, which makes it harder to follow and audit the code, and therefore simplifies the introduction of misleading (malicious) code. Because of this reason, the new smart contract programming language Vyper is giving up on modifiers [29].

The advantages of the pattern are drawn from the fact that it is easy to adapt to different situations and highly reusable, while still providing a secure way to limit the access to functionality and therefore increase smart contract security altogether.

Known Uses

The most prominent example of this pattern is probably the Ownable contract by OpenZeppelin [77]. The research done in section 6.2 shows that roughly 20% of smart contracts deployed since late 2017 inherit from this contract to assign an transferable owner to their contract and only grant access to certain functions to him.

Another example is the core contract of the CrytoKitties DApp[25], where there is not only one owner, but three. Namely the CEO, CFO and COO, who have different security levels and therefore different functions they are allowed to access.

4.2.2 Checks Effects Interactions

Intent

Reduce the attack surface for malicious contracts trying to hijack control flow after an external call.

Motivation

The EVM does not allow for concurrency. When calling an external address, for example, when transferring ether to another account, the calling contract is also transferring the control flow to the external entity. This external entity is now in charge of the control

[25]CryptoKitties core contract: `https://etherscan.io/address/0x06012c8cf97bead5deae237070f9587f8e7a266d#code`

flow and can execute any inherent code, in case it is another contract. Most of the times, this will not cause any problems, but in case the called contract is acting in bad faith, it could alter the control flow and return it in an unexpected state to the initial contract. A possible attack vector is a re-entrancy attack (see subsection 2.4.1 for an example), in which the malicious contract is reentering the initial contract, before the first instance of the function containing the call is finished. This attack can be used to repeatedly invoke functions that should only be executed once and was part of the most prominent hack in Ethereum history: the DAO exploit [3]. The described vulnerability is not present in other software environments, making it hard to avoid for developers not familiar with the quirks of smart contract development. The pattern presented in this section, together with the Secure Ether transfer pattern from subsection 4.2.3, aims to provide a safe solution, in order to make functions unassailable against re-entrancy attacks of any form.

Applicability

Use the Checks Effects Interactions pattern when

- it cannot be avoided to hand over control flow to an external entity.
- you want to guard your functions against re-entrancy attacks.

Participants & Collaborations

Participating entities in this pattern are the called function, as well as the calling party, which will gain the control flow, for example in the case a value transfer is happening. While the pattern is solely implemented in the called function, the other party is a essential participant, as it has the potential to manipulate the control flow.

Implementation

To implement the Check Effects Interactions pattern, we have to be aware of which parts of our function are the susceptible ones. Once we identify that the external call with its insecurities regarding the control flow is the potential cause of vulnerability, we can act accordingly. As stated in the Motivation section of this pattern, a re-entrancy attack can lead to a function being called again, before its first invocation has been finished. We should therefore not make any changes to state variables, after interacting with external entities, as we cannot rely on the execution of any code coming after the interaction. This leaves us with the only option to update all state variables prior to the external interaction. This method can be described as "optimistic accounting" [78], because effects are written down as completed, before they actually took place. For example, the balance of a user will be reduced before the money is actually transferred to him. This is not a problem as will be seen in the Secure Ether Transfer pattern in subsection 4.2.3, because in case something goes wrong with the money transfer, the whole transaction can be reverted, including the reduction of the balance in the state variable. Combined with the Guard Check pattern from subsection 4.1.1, which states that checks should be implemented towards the beginning of a function, we get the natural ordering of: checks first, after that effects to state variables and interactions last.

This method of ordering function components was first described and named in the Solidity documentation [2]. The checks in the beginning assure that the calling entity is in the position to call this particular function (e.g. has enough funds). Afterwards, all specified effects are applied and the state variables are updated. Only after the internal state is fully up to date, external interactions should be carried out. In case this order is followed, a re-entrancy attack should not be able to surpass the checks in the beginning of the function, as the state variables used to check for entrance permission have already been updated.

Sample Code

The following sample code implements a simple banking contract, where users can deposit and withdraw Ether.

```
 1  contract ChecksEffectsInteractions {
 2
 3      mapping(address => uint) balances;
 4
 5      function deposit() public payable {
 6          balances[msg.sender] = msg.value;
 7      }
 8
 9      function withdraw(uint amount) public {
10          require(balances[msg.sender] >= amount);
11
12          balances[msg.sender] -= amount;
13
14          msg.sender.transfer(amount);
15      }
16  }
```

Listing 4.7: Checks Effects Interactions pattern sample code

User balances are stored in a mapping in line 3. The `deposit()` function in line 5 lets the user deposit ether in the contract and stores the respective balances. The actual pattern is implemented in the `withdraw()` function in line 9, which is provided with the amount requested to withdraw. The first step is conducting all necessary **checks**. As this is a small example, there is only one condition to check: if the balance of the user is sufficient for the requested amount, which is done with the help of a `require` statement in line 10. The next step is the application of all **effects**, of which we again have only one: the adjustment of the users balance in line 12. All external **interactions** take place in the last step. We are using the `transfer()` method to further guard the function against re-entrancy, as explained in more detail in the Secure Ether Transfer pattern in subsection 4.2.3.

In an unsafe implementation of this contract, one disregarding the Checks Effects Interactions pattern, where the order of the effect and interaction in line 12 and 14 are exchanged and not `transfer()` but the unsafe and low level `call.value()` is used, a malicious contract could reenter our function. Because the control flow would pass over to the malicious contract, it would be able to call the `withdraw()` function again, before the first invocation is finished, without being intercepted by our check. This is because the line of code carrying the effect of adjusting the balance would not have been reached yet. Therefore, a second transfer of ether to the attacker would be issued. This circle would keep on draining the contract of ether, until either the transaction runs out of gas or the contracts funds are not sufficient anymore.

Consequences

The only negative consequence of using the Checks Effects Interactions pattern is that it is counterintuitive for a developer, being used to a different programming paradigm. In other programming languages it is common procedure to apply effects after the interactions already happened. This is because it is good practice to wait for a return, stating that the function execution was successful, before making any further changes relying on the results.

Once this mental hurdle is overcome, the Checks Effects Interactions pattern is a great way to limit the attack surface of a contract, particularly against re-entrancy attacks, because multiple encapsulated function invocations are not possible anymore. Most of the times it is easy to apply the pattern by only taking the functional code order into account, without having to change any logic. It is a good habit to use this pattern in any function making external calls, regardless of whether the other party is trustworthy or not, because even trusted external parties could transfer control to a third party, which could turn out to be malicious [79].

Known Uses

A short section in the Solidity documentation, as well as the well-known DAO exploit, which showcases the devastating consequences of disregarding its principles, have helped spreading the word about the Checks Effects Interactions pattern. At the time of writing, implementations of it can be observed in a variety of smart contracts. One example is a contract of the CryptoCountries[26] DApp, an interactive game where users can buy and own countries. The `buy(..)` function in line 177 clearly shows the ordering of components: In the beginning several checks are carried out, to assure that the necessary conditions hold true. Only after that the new owner of a country is written to the contract state. At the end of the function the purchase price is transferred to the previous owner.

[26]CryptoCountries ItemRegistry contract: `https://etherscan.io/address/0x17df117bb806a622d841bd5166a23b5d8746232f#code`

The same usage of the pattern was applied in the Own the Day contract[27], an idea similar to the CryptoCountries, only that instead of countries one is trading calendar days this time. The `claimDay(..)` function from line 399 implements the pattern in the same fashion as explained above.

4.2.3 Secure Ether Transfer

Intent

Secure transfer of ether from a contract to another address.

Motivation

Even though the transfer of currency is not the main application of Ethereum, as it is for Bitcoin, it is still a necessary and heavily used feature. While the ether transfer from an external account can simply be done via a network transaction, the transfer of ether from a contract account is not as straight forward.

In the early days of Solidity, the intended way to transfer currency from one contract to another address, whether it is an external account or a contract itself, has been the `<address>.send(amount)` function. It sends the specified amount to the address the function is called on. However, the `send` method does not propagate errors and only sends a small stipend of gas with it. People resorted to workarounds like the `<address>.call.value(amount)()` function, to overcome this limitation, as the amount of gas can be specified, using this approach, by appending `.gas(amountOfGas)` to the `call.value` method. Soon, people discovered that this method opens the door for a new attack vector, not known until then, the re-entrancy attack (see subsection 2.4.1 for an example), which led to a substantial amount of ether being stolen [3]. To give this workaround a name and also include the propagation of exceptions, a new function for the address type was introduced with Solidity version 0.4.13 [80]: `transfer(amount)`. In the end, the intended option to specify the amount of forwarded gas was not implemented for the `transfer` method[28], making it more similar to the `send` function than to the `call.value` function, contrary to what it was supposed to be.

After all, the user is left with three different options to transfer ether from a contract address, each with different attributes and application areas. The aim of this pattern is to delimit the different options from each other and give recommendations on when to use which method, according to the given requirements.

Applicability

Use the Secure Ether Transfer pattern when

[27] Own the Day contract: `https://etherscan.io/address/0x16d790ad4e33725d44741251f100e635c323beb9#code`

[28] And has not been implemented until the time of writing.

- you want to transfer ether from a contract address to another address in a secure way.
- you are not sure which method of ether transfer is the most suitable for your needs.
- you want to guard your contract against re-entrancy attacks.

Participants & Collaborations

The participating entities for this pattern are the contract sending the ether, as well as the address receiving it. The receiving end can either be another contract or an external account. The pattern is implemented at the sending contract. The receiving address, however, also plays a crucial role. Especially if it is another contract, because there is the possibility to reenter the sending contract with malicious intent, in the event that enough gas is forwarded.

Implementation

To make a decision on which method to use in a specific scenario, it is important to first understand the differences in the behavior of the three methods. There are two dimensions to consider when evaluating the characteristics: the **amount of forwarded gas** and **exception propagation**. Both `send` and `transfer` forward a stipend of 2300 gas, which is just enough to log an event at the receiving contract [2]. In case the receiver requires a larger amount of gas to handle the reception of ether, `call.value` has to be used, as it forwards all remaining gas, unless specified otherwise with the help of the `.gas()` parameter. Regarding the propagation of exceptions, `send` and `call.value` are similar to each other, as both of them do not bubble up exceptions but rather return `false` in case of an error. The `transfer()` method, however, propagates every exception that is thrown at the receiving address to the sending contract, leading to an automatic revert of all state changes. Exception propagation for the first two methods can be achieved with a workaround using the Guard Check pattern from subsection 4.1.1. For example: `require(<address>.send(amount))` is equal to `<address>.transfer(amount)`. An overview over the differences of the three methods is given in Table 4.1.

Table 4.1: A comparison of the different ether transfer methods.

Function	Amount of Gas Forwarded	Exception Propagation
send	2300 (not adjustable)	returns `false` on failure
call.value	all remaining gas (adjustable)	returns `false` on failure
transfer	2300 (not adjustable)	throws on failure

There are also similarities between the three functions: All three of them are special functions of the `address` data type, meaning that they have to be appended to an address, where the address is the recipient and the executing contract is the sender. Another similarity is that the amount to be transferred is specified in Wei, which has to be taken into account in order to avoid accounting errors. Finally, it should be pointed out that all three methods are translated into the `CALL` opcode by the Solidity compiler, showing that they share the same internal process.

Sample Code

The following fictitious sample showcases two contracts: the first contract receives ether, while the second one uses the three presented methods to send ether to the first one and shows different ways to handle exception propagation at the same time.

```
 1  contract EtherReceiver {
 2
 3     function () public payable {}
 4  }
 5
 6  contract EtherSender {
 7
 8     EtherReceiver private receiverAdr = new EtherReceiver();
 9
10     function sendEther(uint _amount) public payable {
11         if (!address(receiverAdr).send(_amount)) {
12             //handle failed send
13         }
14     }
15
16     function callValueEther(uint _amount) public payable {
17         require(address(receiverAdr).call.value(_amount).gas(35000)());
18     }
19
20     function transferEther(uint _amount) public payable {
21         address(receiverAdr).transfer(_amount);
22     }
23  }
```

Listing 4.8: Secure Ether Transfer pattern sample code

The only job of the first contract `EtherReceiver` is to receive ether. Therefore, the fallback function carries the `payable` modifier. This fallback function would be the place to implement malicious code in order to carry out a re-entrancy attack. As this attack would require more than the stipend of 2300 gas, only fallback functions triggered by the `call.value` method are able to carry out the attack. The second contract `EtherSender`, starting from line 6, has access to an instance of the `EtherReceiver` in the form of `receiverAdr` (line 8). The instance is used as the target address for the ether transfer.

Each of the three following functions uses one of the three different methods to transfer ether from a contract and carries the `payable` modifier, in order to provide for the ether to be sent in the process. Additionally, each of them takes an amount in Wei to be transferred to the receiver as an input parameter. The amount provided with the transaction should match the amount to be transferred to the receiving contract. The `sendEther` function from line 10 uses the `send` function of the address type. As exceptions for possible errors (e.g. the balance of the contract is lower than the amount to be transferred) are not propagated, we are able to handle the return value, for example in an if-clause.

The second function `callValueEther` from line 16, is encapsulated in a `require` statement to show how exception propagation can be implemented, even for methods not supporting it innately. In this example we specified that the arbitrary amount of 35000 gas should be forwarded to the fallback function of the receiving contract by appending `.gas(35000)` to the method. This can be useful in case the fallback function has some advanced logic implemented.

The `transferEther` function in line 20 has a straight forward implementation and does not need any further statements, as exceptions are automatically propagated.

Consequences

While both methods, `transfer` and `send`, are considered safe against re-entrancy, because they only forward 2300 gas, `transfer` should be the go-to method to transfer ether in most cases. This is because it reverts automatically in case of any errors. The `send` method can be seen as the low level counterpart of `transfer`. It should be used in cases where it is important that the error is handled in the contract without reverting all state changes. The low level `call.value` method should only be used as a last resort, as it breaks the type-safety of Solidity [2]. One of its application fields is sending ether to fallback functions that require more than the stipend of gas. With its adjustable parameters it can provide great flexibility for honest and experienced users, but also for malicious ones.

On one hand, the differentiation into three methods used for the same task allows for flexibility, because the simple `transfer` function can be used for most use cases while the more complicated `call.value` can be adjusted and and used for specialized tasks. On the other hand, the differentiation can be confusing for developers and users alike, as there are no real semantic clues between the naming of the different options, as to where their differences could be.

Another consequence that should be kept in mind are the effects that a limitation on the amount of forwarded gas can have on the surrounding logic. If, for example, a state machine (see subsection 4.1.2) relies on the successful transfer of ether to a specific contract in order to proceed to the next stage, it should be made sure that the recipients are able to receive the ether. Fallback functions requiring more gas than the stipend could otherwise freeze a contract, which is using the `transfer` method. This concept affected the King of the Ether Throne contract, which would have been put into an unresolvable state if it had used `transfer` instead of `send` [81]. To overcome this limitation the Pull Over Push pattern from subsection 4.2.4 can be used, which lets the users request payments instead of proactively sending it, in order to avoid unexpected behavior.

Known Uses

An example of the different possible implementations of this pattern can be seen in this MultiSend contract[29] that lets one send ether to multiple addresses with only one initial transaction. The contract then issues several internal transactions to the recipients on its own and uses either the `transfer` or the `call.value` method, depending on the user's call. Another case is the Crypto Sprites contract[30], a third party game built off of Crypto Kitties. This time only the `transfer` method is used throughout the whole contract, as it is safe and easy to use, and no special functionality was required.

4.2.4 Pull over Push

Intent

Shift the risk associated with transferring ether to the user.

Motivation

Sending ether to another address in Ethereum involves a call to the receiving entity. There are several reasons why this external call could fail. If the receiving address is a contract, it could have a fallback function implemented that simply throws an exception, once it gets called. Another reason for failure is running out of gas. This can happen in cases where a lot of external calls have to be made within one single function call, for example when sending the profits of a bet to multiple winners. Because of these reasons, developers should follow a simple principle: never trust external calls to execute without throwing an error [82]. Most of the times this is not an issue, because it could be argued that it is the responsibility of the receiver to make sure that he/she is able to receive their money, and in case one does not, it is only to his/her disadvantage. The following example code of an auction contract inspired by [79], illustrates how even a single receiver could potentially freeze a whole contract.

```
contract BadAuction {

   address highestBidder;
   uint highestBid;

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

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

```

[29]MultiSend contract: `https://github.com/Alonski/MultiSendEthereum/blob/master/contracts/MultiSend.sol`

[30]Crypto Sprites contract: `https://etherscan.io/address/0xf3C8Ed6C721774C022c530E813a369dFe78a6E85#code`

```
13          highestBidder = msg.sender;
14          highestBid = msg.value;
15      }
16  }
```

Listing 4.9: Auction contract that can potentially be frozen

Once an address which is not able to receive ether via the `transfer` method (for example because the fallback function requires more than the forwarded gas; see subsection 4.2.3 for more detailed information) takes the position of the highest bidder, the contract will be in an unsolvable state. Every new attempt to outbid the current highest bidder will trigger the ether transfer in line 10, which will result in an exception and therefore make it impossible to overbid the current leader.

Another potential problem arises when trying to send ether to multiple recipients with one function call. It only needs one of the transfers to fail in order to revert all transfers that already happened and stop the following transfers from executing.

To overcome these limitations a technique has been proposed [79, 82] that isolates each external call and shifts the risk of failure from the contract to the user. Due to the isolation of the transfers, no other transfers or contract logic have to rely on its successful execution.

Applicability

Use the Pull over Push pattern when

- you want to handle multiple ether transfers with one function call.
- you want to avoid taking the risk associated with ether transfers.
- there is an incentive for your users to handle ether withdrawal on their own.

Participants & Collaborations

The Pull over Push pattern consists of three participants. First, the entity responsible for the initiation of the transfer (e.g. the owner of a contract, or the contract itself) starts the process. Secondly, the smart contract has the responsibility of keeping track of all balances. The third participant is the receiver, who will not simply receive his funds via a transaction, but has to actively request a withdrawal, in order to isolate the process from other payout and contract logic.

Implementation

In order to isolate all external calls from each other and the contract logic, the Pull over Push pattern shifts the risk associated with the ether transfer to the user, by letting him withdraw (**pull**) a certain amount, which would otherwise have to be sent to him (**push**). A core component of this implementation is a mapping, which keeps track of the outstanding balances of the users. Instead of performing an actual ether transfer from the contract to a recipient, a function is called, which adds an entry to the mapping, stating that the user is eligible to withdraw the specified amount. In case the mapping already contains an entry for this address, the amount is added to the existing one. The user is now responsible to withdraw the funds by issuing a transaction to a withdrawal method of the smart contract that uses the Checks Effects Interactions pattern from subsection 4.2.2 to update the outstanding balance before actually transferring the ether.

Implemented this way, a thrown exception in one of the transfers would only affect this specific transfer and not a whole series of transfers or even the whole contract, like in Listing 4.9.

Sample Code

An exemplary implementation of the Pull over Push pattern can be seen in the following code, which contains only the necessary components.

```
 1  contract PullOverPush {
 2
 3      mapping(address => uint) credits;
 4
 5      function allowForPull(address receiver, uint amount) private {
 6          credits[receiver] += amount;
 7      }
 8
 9      function withdrawCredits() public {
10          uint amount = credits[msg.sender];
11
12          require(amount != 0);
13          require(address(this).balance >= amount);
14
15          credits[msg.sender] = 0;
16
17          msg.sender.transfer(amount);
18      }
19  }
```

Listing 4.10: Pull over Push pattern sample code

The `credits` mapping in line 3 is one of the key elements of this pattern and stores the amount of ether (in Wei) that each address is allowed to withdraw. The permission for withdrawal happens in the `allowForPull(..)` function in line 5. This function should be used instead of every ether transfer that is supposed to be settled with a pull instead of a push payment. So instead of `<address>.transfer(amount)`, we would now use `allowForPull(<address>, amount)`. The function carries the `private` keyword and can therefore only be called from within the contract. In case the pull permissions should be given directly from the outside via a transaction, the function can be made `public`. The Access Restriction pattern from subsection 4.2.1 should be used in that case, to make sure that only authorized addresses can issue withdrawal credits.

To request a withdrawal the eligible users have to call the `withdrawCredits()` function from line 9. Line 10 stores the amount the caller is allowed to withdraw in memory. Afterwards, line 12 makes sure that the requesting user has been credited an amount to withdraw higher than zero[31]. Line 13 requires the contract balance to be high enough to cover the requested amount. An exception would be thrown at the actual transfer later on anyways, if this condition was violated, so this check is not absolutely necessary. However, it is good practice to fail as early as possible [82]. Line 15 sets the allowed withdraw amount in the mapping to zero, before actually transferring it, in order to be conform with the Checks Effects Interactions pattern and avoid re-entrancy. At last, the amount is transferred to the recipient in line 17 via a push.

Consequences

The use of the Pull over Push pattern is a good way to mitigate some of the quirks that come with Solidity when sending ether, especially when performing multiple transfers at once. Due to the isolation of the error prone transfer functionality, one failed transfer does not lead to a revert of all successful operations anymore. Additionally, it is now the responsibility of the requesting user to make sure that he is able to receive ether.

However, the negative consequences, coming with the additional steps required, should not be ignored. Interacting with a contract that uses pull instead of push payments requires the users to send one additional transaction, namely the one requesting the withdrawal. This does not only lead to higher gas requirements and therefore higher transaction costs, but also harms the user experience as a whole. Users should not have to interact more with a smart contract than absolutely necessary, as users, especially inexperienced ones, tend to make mistakes. In one case, a smart contract owner reported that more than 10% of the users did not withdraw their funds from the contract in the seven days they were given, before he could have collected the remaining contract balance for himself [83]. This implies that this pattern should only be used if there is a strong incentive for all participants to withdraw their funds. Otherwise, users might consider a competitor or not using the contract at all, if withdrawing is to complicated or simply not worthwhile.

Using this pattern can be considered a trade-off between security and convenience for the users. Before implementing it, one should evaluate if the cutback in user experience is manageable, or if a clever use of the Secure Ether Transfer pattern from subsection 4.2.3 might be sufficient to rule out any vulnerabilities.

Known Uses

One popular example of using the Pull over Push pattern is the PullPayment contract by OpenZeppelin[32]. The contract implements the pattern in a general way and contracts

[31]Since an unsigned integer cannot be negative, it is sufficient to check that the amount is not equal zero.
[32]PullPayment contract: `https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/payment/PullPayment.sol`

wanting to use its functionality can inherit from it.

A more specialized implementation can be found in a contract called BlockParty[33], a contract to manage attendance deposits for free events. Users are only getting their deposit back if they showed up at the event they registered for. Attendants are able to request a withdrawal after the contract owner, in most cases the organizer of the event, has confirmed their attendance via a transaction containing their Ethereum address.

4.2.5 Emergency Stop

Intent

Add an option to disable critical contract functionality in case of an emergency.

Motivation

Even heavily audited and tested code may contain bugs or defective code segments. Smart contracts are no exception to this. Oftentimes these bugs do not get discovered until they are used for an attack by an adversary. Once a critical flaw is discovered, it is hard to fix, because immutability is one of the core principles of the blockchain. While several patterns allow for upgradeable code to a certain degree (see section 4.3), these solutions usually take a substantial amount of time to implement and come into play. During this window of time, the attackers could continue with their hack, possibly draining all available funds from the contract before the fix is broadcast to the network.

With the help of this pattern, we provide the possibility to pause a contract by blocking calls of critical functions, preventing attackers from continuing their work. Of course, this pattern can be used to prevent the exploit of any kind of bug, regardless if it was discovered by an attacker or a benign entity, until the smart contract is fixed, or other countermeasures have been taken.

Applicability

Use the Emergency Stop pattern when

- you want to have the ability to pause your contract.
- you want to guard critical functionality against the abuse of undiscovered bugs.
- you want to prepare your contract for potential failures.

Participants & Collaborations

There are three major participants in this pattern: The central component is a state variable that indicates, if the contract is currently stopped or not. This variable is

[33]BlockParty contract: `https://github.com/makoto/blockparty/blob/master/contracts/Conference.sol`

referenced in the individual functions that are either not accessible, or only accessible, while the contract is stopped. The third participant consists of the group of entities that have the clearance to issue the emergency stop. This could for example be the contract owner or a certain majority of users.

Implementation

The state of the contract being currently stopped or not is stored in a state variable in the form of a Boolean, which is initialized as `false` during contract creation. To stop the contract in case of an emergency, this state variable has to be set to `true`. The proposed way to do this is via a function call. To avoid the exploitation of the stopping functionality by random persons, only authorized entities (e.g. the contract owner) should be able to invoke this function. The Access Restriction pattern from subsection 4.2.1 can be used for this task. Another option to prevent misuse, while keeping up the notion of decentralization at the same time, would be to implement a rule set that has to be fulfilled in order to trigger the stopping mechanism. A variety of possible rules could be applied, depending on the respective use case (e.g. 10% of the contracts balance have been withdrawn in the last hour).

Once the state variable can be activated by setting it to `true`, we can again use the Access Restriction pattern to make sure that the functions identified as critical cannot be called anymore, as soon as the contract is stopped. This is achieved with the help of a function modifier that throws an exception, in case the state variable indicates that emergency stop has been triggered. Functions that should be available during the stop, because they can help resolve the situation, like letting users withdraw their deposits, can be made available in the same way.

Another design decision dependent on the use case of the contract is, if the emergency stop should be revertible or not. In case the contract is supposed to be resumable, for example because precautions for upgradability have been taken, the resumption of the contract is initiated by setting the state variable that indicates the stopping status of the contract, back to `false`. The function for this task can be implemented in the same way as the function to initiate the emergency stop and should also be guarded against unauthorized calls.

Sample Code

The concrete implementation of this pattern is highly dependent on the context of the underlying smart contract. Important issues, like the recovery after an emergency stop, or which functions to make available and which not, should be assessed and carefully tested before deploying the contract.

The following code shows the basic framework of the Emergency Stop pattern and provides two exemplary methods that are influenced by a stop. For the sake of clarity, any further contract logic is omitted.

```
1  contract EmergencyStop {
2  
3      bool isStopped = false;
4  
5      modifier stoppedInEmergency {
6          require(!isStopped);
7          _;
8      }
9  
10     modifier onlyWhenStopped {
11         require(isStopped);
12         _;
13     }
14 
15     modifier onlyAuthorized {
16         // Check for authorization of msg.sender here
17         _;
18     }
19 
20     function stopContract() public onlyAuthorized {
21         isStopped = true;
22     }
23 
24     function resumeContract() public onlyAuthorized {
25         isStopped = false;
26     }
27 
28     function deposit() public payable stoppedInEmergency {
29         // Deposit logic happening here
30     }
31 
32     function emergencyWithdraw() public onlyWhenStopped {
33         // Emergency withdraw happening here
34     }
35 }
```

Listing 4.11: Emergency Stop pattern sample code

The Boolean `isStopped` in line 3 is the state variable carrying the information, if the contract is currently stopped or not. This variable is checked by the two modifiers `stoppedInEmergency` (line 5) and `onlyWhenStopped` (line 10) in order to restrict access to the functions utilizing them. The third modifier `onlyAuthorized` in line 15 checks, if the caller of a method has authorization to do so. This could for example be restricted to the owner by adding `require(msg.sender == owner)`, or a voting mechanism [79]. The two functions in line 20 and 24 use this modifier and allow an authorized caller to stop and resume the contract by setting the state variable `isStopped` to either `true` or `false` respectively.

The `deposit()` function from line 28 is an example for a critical method that should be inaccessible once the contract is stopped. This restriction is achieved by appending the `stoppedInEmergency` modifier to the function header. An exception would be thrown, if the function would be called during a stop. The `emergencyWithdraw()` function in line 32 works the other way around. It is only accessible during the emergency stop, due to the `onlyWhenStopped` modifier. An emergency function like this should be implemented to give contract users an option to access their funds during a shutdown. Otherwise, users would have to trust the contract owner not to arbitrarily freeze their funds without the possibility to get them back.

Consequences

Applying the Emergency Stop pattern to a contract adds a fast and reliable method to halt any sensitive contract functionality, as soon as a bug or another security issue is discovered. This leaves enough time to weigh all options and possibly upgrade the contract in order to fix the security breach.

The negative consequence of having an emergency stop mechanism from a users point of view is, that it adds unpredictable contract behavior. Unless a cogent rule set has to be fulfilled before triggering the stopping mechanism is possible, there is always the possibility that the stop is abused in a malicious way by the authorized entity. Therefore, the emergency stop implemented by this pattern, should only be triggered as a last resort, and not be seen as a pausing mechanism for predictable events. The State Machine pattern with timed transitions from subsection 4.1.2, can be used for such cases in order to keep contract behavior predictable for users and minimize the trust that has to be put into the system.

Known Uses

The most prevalent application of the Emergency Stop pattern is the Pausable contract[34] from the OpenZeppelin library. This straightforward contract is implementing the pattern and every contract wanting to use it, can inherit from it. Several applications could be observed making use of this technique: one example is OmiseGO[35], a token with the aim to enable financial inclusion and interoperability through a decentralized network. The contract utilizes an older version of the Pausable contract starting from line 274.

Another, though less common, possibility is implementing the pattern on your own. One example for this approach is the contract of the Million Ether Homepage[36]. In this case the Emergency Stop pattern is implemented inside the main contract and gives the owner of the contract the ability to stop execution of several functions at any given time.

[34]Pausable contract: `https://github.com/OpenZeppelin/zeppelin-solidity/blob/master/contracts/lifecycle/Pausable.sol`

[35]OmiseGO contract: `https://etherscan.io/address/0xd26114cd6EE289AccF82350c8d8487fedB8A0C07#code`

[36]MillionEtherHomepage contract: `https://etherscan.io/address/0x15dbdB25f870f21eaf9105e68e249E0426DaE916#code`

4.3 Upgradeability Patterns

Smart contracts on the Ethereum blockchain are immutable. While immutability is one of the key attributes that make blockchains interesting for execution of smart contracts, it also creates problems regarding the maintainability of contracts on the network. The discovery of bugs, the need to upgrade business logic, or adjustments that have to be made because of changes to the EVM, are just some of the reasons a contract has to be updated. An example, for the need of upgradeable code is the recent Parity wallet incident [4], where Ether worth a total of USD260 million were frozen. The defective contract[37] contains a constant reference to a library in line 451: `address constant _walletLibrary = 0x863df6bfa4469f3ead0be8f9f2aae51c91a907b4;`, with no way of changing it. Due to a bug in the code, a user was able to delete the referenced library, freezing all funds relying on the, now unreachable, functionality. Had the address of the library been updateable, the problem could have been solved with one single function call.

Immutability and upgradeability are two concepts that are notoriously hard to unite. This is especially true for software developers being used to a different environment. For example in web development, code is stored centrally on a server. It is possible to push a bug fix in the matter of minutes, since only one instance of code has to be updated. This method is not possible in the context of a blockchain, due to immutability of code and decentralized storage. However, as far back as the beginning of Ethereum, the Ethereum white paper made a proposal for how to overcome this limitation:

> Although code is theoretically immutable, one can easily get around this and have de-facto mutability by having chunks of the code in separate contracts, and having the address of which contracts to call stored in the modifiable storage [9].

It is suggested to split up code into distinct modules and make use of contract storage, which can be altered to store references to the respective contracts. To upgrade a smart contract, the new version of the contract is published to the blockchain and the old address in storage is replaced with the new address. Entities wanting to interact with the contract can look up the address of the current version in storage. This general concept can be realized in several ways. Tanner [84] gives an overview over several possible methods that can be used to prepare smart contracts for future upgrades and lists the research that lead to these findings. The work by Marino and Juels [85] also revolves around altering smart contracts, but with a focus on modification of contracts due to legal reasons.

The two patterns presented in this category are supposed to be a fundamental framework for the entry into upgradeable smart contracts. They provide solutions for calling distinct modules via a proxy and the separation of contract storage from logic.

[37]Defective Parity wallet contract: `https://etherscan.io/address/0x3bfc20f0b9afcace800d73d2191166ff16540258#code`

4.3.1 Proxy Delegate

Intent

Introduce the possibility to upgrade smart contracts without breaking any dependencies.

Motivation

As explained in the overview of this section, mutability in Ethereum is hard to achieve, but necessary. It allows developers to adapt to a changing environment and to react to bugs and other errors. To overcome the limitations introduced by the immutability of contract code, a contract can be split up into modules, which are then virtually upgradeable. We use the term "virtually", as existing contracts still cannot be changed. However, a new version of the contract can be deployed and its address replaces the old one in storage. To avoid breaking dependencies of other contracts that are referencing the upgraded contract or users, who do not know about the release of a new contract version (that comes with a new address), we make use of a proxy[38] contract that delegates calls to the specific modules. These modules are also called delegates, as work is delegated to them by the proxy. A first functional version of this pattern was introduced in [86].

This example makes use of a special message call, named `delegatecall`[39] [87]. Using this new message call allows a contract to pass on the function call to the delegate without having to explicitly know the function signature, a crucial point for upgradeability. Another difference to a regular message call is, that with a `delegatecall` the code at the target address is executed in the context of the calling contract [2]. This means that the storage and state of the calling contract are used. Additionally, transaction properties like `msg.sender` and `msg.value` will remain the ones of the initial caller.

This pattern often goes hand in hand with the Eternal Storage pattern described in subsection 4.3.2 to further decouple storage from contract logic.

Applicability

Use the Proxy Delegate pattern when

- you want to delegate function calls to other contracts.
- you need upgradeable delegates, without breaking dependencies.
- you are familiar with advanced concepts like delegatecalls and inline assembly.

Participants & Collaborations

There are several participants interacting with each other in this pattern. The basic idea is that a caller (external or contract address) makes a function call to the proxy, which

[38]Sometimes also called dispatcher.

[39]The `DELEGATECALL` opcode had just been released with the Homestead release (14.03.16) at the time.

delegates the call to the delegate, where the function code is located. The result is then returned to the proxy, which forwards it to the caller. To know at which address the current version of the delegate resides, the proxy can either store it itself in a variable or in case the Eternal Storage pattern from subsection 4.3.2 is used, consult the external storage for the current address.

Because `delegatecall` is used to delegate the call, the called function is executed in the context of the proxy. This further means that the storage of the proxy is used for function execution, which results in the limitation that the storage of the delegate contract has to be append only. What this means is, that in case of an upgrade, existing storage variables cannot be omitted or changed, only new variables are allowed to be added [86]. This is because changing the storage structure in the delegate would mess up storage in the proxy, which is expecting the previous structure. An example for this behavior can be found in the GitHub repository[40].

Implementation

The implementation of the Proxy part of the pattern is more complex than most of the other patterns presented in this thesis. A `delegatecall` is used to execute functions at a delegate in the context of the proxy and without having to know the function identifiers, because `delegatecall` forwards the `msg.data`, containing the function identifier in the first four bytes. In order to trigger the forwarding mechanism for every function call, it is placed in the proxy contract's fallback function. Unfortunately a `delegatecall` only returns a boolean variable, signaling the execution outcome. When using the call in the context of a proxy, however, we are interested in the actual return value of the function call. To overcome this limitation, inline assembly[41] has to be used. With the help of inline assembly we are able to dissect the return value of the `delegatecall` and return the actual result to the caller. Due to the complexity of inline assembly, any further explanation on the implemented functionality will be done with the help of an example in the Sample Code section of this pattern. One way of circumventing the need for inline assembly would be returning the result to the caller via events. While events cannot be accessed from within a contract, it would be possible to listen to them from the front end and act according to the result from there on. This method, however, will not be discussed in this pattern.

As stated in the Participants & Collaborations section, the upgrading mechanism, hence, storing of the current version of the delegate, can either happen in external storage or in the proxy itself. In case the address is stored in the proxy, a guarded function has to be implemented, which lets an authorized address update the delegate address.

The delegate can be implemented in the same way as any regular contract and no

[40]Overwriting storage example: `https://github.com/fravoll/solidity-patterns/blob/master/ProxyDelegate/StorageOverwriteExample.sol`

[41]Inline assembly allows for more precise control over the stack machine, with a language similar to the one used by the EVM and can be used within solidity code [2].

special precautions have to be taken, as the delegate does not have to know about the proxy using its code. The only thing that has to be taken into account is, that while upgrading the contract, storage sequence has to be the same; only additions are permitted.

Sample Code

This generic example of a Proxy contract is inspired by [88] and stores the current version of the delegate in its own storage. Because the design of the Delegate contract can take many forms, there is no explicit example given.

```
1  contract Proxy {
2
3      address delegate;
4      address owner = msg.sender;
5
6      function upgradeDelegate(address newDelegateAddress) public {
7          require(msg.sender == owner);
8          delegate = newDelegateAddress;
9      }
10
11     function() external payable {
12         assembly {
13             let _target := sload(0)
14             calldatacopy(0x0, 0x0, calldatasize)
15             let result := delegatecall(gas, _target, 0x0, calldatasize, 0x0, 0)
16             returndatacopy(0x0, 0x0, returndatasize)
17             switch result case 0 {revert(0, 0)} default {return (0, returndatasize)}
18         }
19     }
20 }
```

Listing 4.12: Proxy Delegate pattern sample code

The address variables in line 3 and 4 store the address of the delegate and the owner, respectively. The `upgradeDelegate(..)` function is the mechanism that allows a new version of the delegate being used, without the caller of the proxy having to worry about it. An authorized entity, in this case the owner (checked with a simple form of the Access Restriction pattern from subsection 4.2.1 in line 7) is able to provide the address of a new delegate version, which replaces the old one (line 8).

The actual forwarding functionality is implemented in the function starting from line 11. The function does not have a name and is therefore the fallback function, which is being called for every unknown function identifier. Therefore, every function call to the proxy (besides the ones to `upgradeDelegate(..)`) will trigger the fallback function and execute the following inline assembly code: Line 13 loads the first variable in storage, in this case the address of the delegate, and stores it in the memory variable `_target`. Line 14 copies the function signature and any parameters into memory. In line 15 the `delegatecall` to the `_target` address is made, including the function data that has been stored in memory. A boolean containing the execution outcome is returned and stored in the `result` variable. Line 16 copies the actual return value into memory. The switch in line 17 checks whether the execution outcome was negative, in which case any state changes are reverted, or positive, in which case the result is returned to the caller of the proxy.

Consequences

There are several implications that should be considered when using the Proxy Delegate pattern for achieving upgradeability. With its implementation, complexity is increased drastically and especially developers, who are new to smart contract development with Solidity, might find it difficult to understand the concepts of delegatecalls and inline assembly. This increases the chance of introducing bugs or other unintended behavior. Another point are the limitations on storage changes: fields cannot be deleted nor rearranged. While this is not an insurmountable problem, it is important to be aware of, in order to not accidentally break contract storage. An important negative consequence from a social perspecive is the potential loss in trust from users. With upgradeable contracts, immutability as one of the key benefits of blockchains, can be avoided. Users have to trust the responsible entities to not introduce any unwanted functionality with one of their upgrades. A solution to this caveat could be strategies that only allow partial upgrades. Core features could be non-upgradeable, while other, less essential, features are implemented with the option for upgrades [84]. If this approach is not applicable, a trust loss could also be mitigated by introducing a test period, during which upgrades can be carried out. After the expiration of the test period, the contract cannot be changed any longer.

Besides these negative consequences, the Proxy Delegate pattern is an efficient way to separate the upgrading mechanism from contract design. It allows upgradeability without breaking any dependencies.

Known Uses

Implementations of the Proxy Delegate pattern are more likely to be found in bigger DApps, containing a large number of contracts. One example for this is Augur[42], a prediction market that lets users bet on the outcome of future events. Another example is the EtherRouter contract of Colony[43], which is a platform for creating decentralized organizations. In both cases, Augur and Colony, the address of the upgradeable contract is not stored in the proxy itself, but in some kind of address resolver.

[42]Augur Delegator contract: `https://github.com/AugurProject/augur-core/blob/master/source/contracts/libraries/Delegator.sol`

[43]Colony EtherRouter contract: `https://github.com/JoinColony/colonyNetwork/blob/develop/contracts/EtherRouter.sol`

4.3.2 Eternal Storage

Intent

Keep contract storage after a smart contract upgrade.

Motivation

When upgrading a smart contract, e.g. with the help of the Proxy Delegate pattern from subsection 4.3.1, what really happens is that a new version of the contract is deployed to the network and coexists with the old version. Because the old contract is not actually updated to a new version, the accumulated storage still resides at the old address. This usually includes important data, like user information, account balances or references to other contracts, which are still needed in the new version of the contract. One option would be to implement functionality to read every item from the old storage and store it in the new one. There are at least two issues with this approach: Firstly, writing to storage is one of the most expensive operations in Ethereum [11]. Successively reading every single storage entry and storing it at the new address, every time a contract is upgraded, would be unreasonable from an economic point of view. There would even be a chance that the transaction carrying out the storage migration would run out of gas, in case there are too many entries to store. Secondly, the whole storage migration would need to be already planned during creation time and a lot of additional logic would need to be included to carry out the migration.

A more practical solution was first proposed in [89]. It solves the problems of migrating storage by separating the storage from contract logic. A separate contract, with the sole purpose of acting as a storage to another contract, is introduced. It should be as flexible as possible, in order to avoid the need for an upgrade of the storage structure, as this would introduce the same problems as explained before. The storage is supposed to last over the whole lifetime of the initial contract, therefore the name eternal storage. A new version of the smart contract can simply use the same storage contract as its predecessor, after it has been registered.

Applicability

Use the Eternal Storage pattern when

- your contract is upgradeable and should retain storage after an upgrade.
- you want to avoid problems with migrating storage after a contract upgrade.
- you can accept a slightly more complex syntax for storing data.

Participants & Collaborations

There are three entities involved in this pattern. The central point is the smart contract implementing the Eternal Storage pattern. It provides its storage for another contract.

The administrative work is done by an owner, which could be the person responsible for the DApp, or an autonomous organization. The administrator can set the address of the latest version of the contract using the storage, and update it once the address has changed due to an upgrade. The last remaining entity is the contract in need of the storage. This is the contract at the address set by the administrator, and has the authorization to send and retrieve elements from the storage contract.

Implementation

It would be possible to implement this pattern with a rigid representation of the needed storage, by implementing only the currently used data tapes in the eternal storage. To avoid upgrades to the data store, however, it should be designed as flexible as possible. This flexibility is achieved by implementing several mappings, one for each data type, in which data can be stored. These mappings map the abstracted-down value to a certain SHA-3 hash, acting as a key-value storage. A SHA-3 hash is used as the key, in order to allow identifiers of arbitrary length to be used as keys. Using hashes as keys also enables the storage of complicated data types, like mappings (e.g. using `keccak256("balances", "UserID123")` as the key for storing the balance of the user with the ID 123).

Each mapping should be equipped with three functions to manage storage, retrieval and deletion. The storage function saves a provided value and the associated key in the respective mapping, depending on the data type of the value. The retrieval function returns the value for a provided key. To delete an existing entry in a mapping, the deletion function is called with the key of the item to be deleted as an input parameter. Because the functions for storage and deletion are affecting the contract state, they should be guarded by the Access Restriction pattern from subsection 4.2.1, so access is only allowed from the most recent version of the contract using the eternal storage. The address of the current version can be stored in a variable and should only be changeable by authorized addresses.

The hashing of the storage key should take place at the calling contract in order to have a uniform function interface for the eternal storage that always expects keys of the `bytes32` data type. Since the proposed approach is more complex than regular value storing, wrappers can help to reduce complexity and make code more readable [90].

Sample Code

The following sample code showcases a possible implementation of the Eternal Storage pattern and is inspired by the implementation in [91]. For the sake of space, this implementation only features the two data types `uint` and `address`. The remaining data types are implemented accordingly. The full implementation with a total of six data types can be found in the GitHub repository[44].

[44]Eternal Storage full example: `https://github.com/fravoll/solidity-patterns/blob/master/EternalStorage/EternalStorage.sol`

```
 1  contract EternalStorage {
 2
 3      address owner = msg.sender;
 4      address latestVersion;
 5
 6      mapping(bytes32 => uint) uIntStorage;
 7      mapping(bytes32 => address) addressStorage;
 8
 9      modifier onlyLatestVersion() {
10         require(msg.sender == latestVersion);
11          _;
12      }
13
14      function upgradeVersion(address _newVersion) public {
15          require(msg.sender == owner);
16          latestVersion = _newVersion;
17      }
18
19      // *** Getter Methods ***
20      function getUint(bytes32 _key) external view returns(uint) {
21          return uIntStorage[_key];
22      }
23
24      function getAddress(bytes32 _key) external view returns(address) {
25          return addressStorage[_key];
26      }
27
28      // *** Setter Methods ***
29      function setUint(bytes32 _key, uint _value) onlyLatestVersion external {
30          uIntStorage[_key] = _value;
31      }
32
33      function setAddress(bytes32 _key, address _value) onlyLatestVersion external {
34          addressStorage[_key] = _value;
35      }
36
37      // *** Delete Methods ***
38      function deleteUint(bytes32 _key) onlyLatestVersion external {
39          delete uIntStorage[_key];
40      }
41
42      function deleteAddress(bytes32 _key) onlyLatestVersion external {
43          delete addressStorage[_key];
44      }
45  }
```

Listing 4.13: Eternal Storage pattern short sample code

In line 3 the variable `owner` is initialized with `msg.sender` to specify an authorized entity. Administration could be implemented in several other ways, for example to authorize several addresses, or let an autonomous organization take the role of the owner. Line 4 stores the current address of the latest version of the contract using this storage. The mappings in line 6 and 7 are acting as the key-value store, where the actual data is stored. The `onlyLatestVersion()` modifier from line 9 makes sure, that only the address of the latest version is able to insert and delete values from storage. The `upgradeVersion(..)` function in line 14 is only callable by the owner[45] and lets him update the address of the latest contract version.

The two functions from line 20 onward are the getters and return the value for a provided key. The two following functions in line 29 and 33 are setters that take a hash and a value as input parameters and store them in the respective mapping. The last two functions in line 38 and 42 are responsible for deleting entries from the mappings when provided with a key. These two functions, as well as the two setters right before, are only callable by the latest version of the contract, because they make use of the `onlyLatestVersion()` modifier.

The upgradeable contract that uses the eternal storage, can implement wrappers to facilitate dealing with the unfamiliar syntax using hashes as keys. The following code shows three exemplary wrappers to help manage user balances.

```
function getBalance(address balanceHolder) public view returns(uint) {
    return eternalStorageAdr.getUint(keccak256("balances", balanceHolder));
}

function setBalance(address balanceHolder, uint amount) internal {
    eternalStorageAdr.setUint(keccak256("balances", balanceHolder), amount);
}

function addBalance(address balanceHolder, uint amount) internal {
    setBalance(balanceHolder, getBalance(balanceHolder) + amount);
}
```

Listing 4.14: Eternal Storage example wrappers code

The `getBalance(..)` function retrieves the balance for a key, which is generated as the hash of the string `"balances"` combined with the address of the balance holder. The `setBalance(..)` function works in a similar way, only setting a balance instead of getting it. In both functions a combination of variables is hashed in order to generate a unique key that can be used to access the mappings. The last function `addBalance(..)` makes use of the previous two by first getting the balance of a user, then adding an amount to it and storing it again in the end. It is a good example for how wrappers can help reduce complexity, for example by concealing the hashing mechanism.

Consequences

The obvious advantage of the Eternal Storage pattern is the elimination of the need for storage migration after upgrading a smart contract. A newly deployed contract version can call the same storage contract that its predecessor used, after it has been registered.

[45]This is achieved with a simple `require` statement instead of a modifier, because it is the only function in the contract that should only be callable by the owner.

It can read from it or store new key-value pairs. Positive consequences specific to the proposed approach, with the use of hashes as keys in the key-value store, revolve around flexibility. The eternal storage is flexible, because virtually every data type can be stored. That in turn makes the eternal storage flexible against any possible changes in the data scheme of the calling contract, without having to upgrade it.

Several negative consequences have to be considered before implementing this pattern as a storage solution. The separation of logic and storage increases complexity, because external calls have to be made. External calls should always be handled with caution, as they can cause unintended behavior [79]. Further complexity is introduced by additional syntax, including the need for hash functions, which can be mitigated to some extend by the use of wrappers. This additional complexity is also reflected in higher gas consumption. However, an internal test suggests, that gas requirements would only increase by about 5%, as the additional computation costs are only marginally relevant, compared to expensive storage operations. As already addressed in the Proxy Delegate pattern in subsection 4.3.1, the bypass of immutability can influence the trust users are willing to put into a DApp. Depending on the exact implementation, an authorized entity could be able to set a malicious contract as the latest version and alter the storage to his advantage. This issue can be mitigated by a proper rule set for version changes or decentralized ownership.

Known Uses

The Eternal Storage pattern is used in the upgradeability strategy of Rocket Pool[46], a decentralized proof of stake pool. They try to overcome the issue of trust loss by disabling the direct access of the owner to the contract, after he has initialized it.

4.4 Economic Patterns

The need for reduced gas requirements is getting more and more important as the network grows. Even though the gas price, measured in Wei, has been fairly constant over the last two years, the price of Wei in USD has risen by over 1000% in 2017 [92] and could get even higher in the future. This means that transactions are getting more and more expensive (if valued in USD) as the price of Ether is increasing. A gas-efficient contract can therefore be a competitive advantage. Users having the choice between several similar alternatives might be inclined to chose the contract that is the cheapest to interact with, as they are the ones paying for the required gas.

Considering these points, it would only be logical for a developer to counter the development of increasing transaction costs by writing their code in a way that minimizes the required gas from the beginning. However, Chen et al. identified 7 anti-patterns that make a contract require more gas than necessary [93]. They found out that over 80% of

[46]Rocket Pool Storage contract: `https://github.com/rocket-pool/rocketpool/blob/master/contracts/RocketStorage.sol`

the contracts they observed implemented one of the patterns. These findings show, that resourceful programming is not yet applied at a large scale for smart contracts. Growing adoption of Ethereum would lead to more transactions and growing prices for Ether in the future, which would directly influence the transaction costs. In such a scenario, the possibility to reduce transaction costs without limiting the functionality, would become even more important.

We therefore present three patterns in this section, that share one characteristic: they all reduce the amount of gas needed when interacting with the implementing contract. Unlike the patterns from other categories, ease of use, predictable behavior or security are not the main criteria when evaluating these patterns. While all the patterns in this category are safe to use and interference in contract logic is kept to a minimum, it is possible that readability is suffering from the way logic has to be implemented when reducing gas requirements.

One key difference to the way we presented the previous patterns is an additional layer of description called *Gas Analysis*. It is used to show the difference in required gas without and with the implementation of the pattern by providing empirical data on gas usage. The data was collected using the Solidity online compiler Remix[47]. The code for the experiments can be found on GitHub[48].

4.4.1 String Equality Comparison

Intent

Check for the equality of two provided strings in a way that minimizes average gas consumption for a large number of different inputs.

Motivation

Comparing strings in other programming languages is a trivial task. Built-in methods or packages can check for the equality of two inputs in one single call, e.g. `String1.equals(String2)` in Java. Solidity does not support any functionality like this at the time of writing. Therefore, we provide a reliable and gas efficient pattern to check, if two strings are equal or not.

Several solutions to this problem have been implemented over the last years. One of the first was part of the StringUtils library[49] provided by the Ethereum Foundation, which did a pairwise comparison of each character and returned false as soon as one pair did not match. This solution returns correct results and uses little gas for short strings and cases where the difference in characters occurs early on. However, gas consumption can get very high for strings that are actually equal as well as long pairs, where the difference is not already in the first few characters. This is because the

[47]Remix: `https://remix.ethereum.org/`
[48]Experiment Code: `https://github.com/fravoll/solidity-patterns`
[49]StringUtils library: `https://github.com/ethereum/dapp-bin/blob/master/library/stringUtils.sol`

algorithm has to do a lot of comparisons in these cases. Other forces that lead to varying gas requirements, are differences in the average length of strings, as well as different probabilities of correctness. Highly variable and unpredictable gas requirements are a problem for smart contracts, as they bear the risk for transactions to run out of gas and lead to unintended behavior. Therefore, a low, stable and predictable gas requirement is desired.

The solution we propose to mitigate the problem of scaling gas requirement is the usage of a hash function for comparison, combined with a check for matching length of the provided strings, to weed out pairs with different lengths from the start.

Applicability

Use the String Equality Comparison pattern when

- you want to check two strings for equality.
- most of your strings to compare are longer than two characters.
- you want to minimize the average amount of gas needed for a broad variety of strings.

Participants & Collaborations

As there are easier methods to compare two strings than on a blockchain, this pattern is intended mainly for internal use in smart contracts as well as in libraries. In both cases there are only two participants, the called function, which implements the pattern and conducts the actual comparison as well as a calling function. The calling function can call from within the same contract or an inheriting one, or from an external contract, in case of usage in a library.

Implementation

The implementation of this pattern can be grouped into two parts:

- The first step checks if the two provided strings are of the same length. If this is not the case, the function can return that the two strings are not equal and the second step is therefore skipped. To compare the length, the strings have to be converted to the `bytes` data type, which provides a built-in length member. This first step is needed to sort out any string pairs with different length and safe the gas for the hash functions in these cases.
- In the second step, each string is hashed with the built-in cryptographic function `keccak256()` that computes the Keccak-256 hash of its input [2]. The calculated hashes can then be compared and prove, in case of a complete match, that the two inputs are equal to each other.

Sample Code

```
1 function hashCompareWithLengthCheck(string a, string b) internal returns (bool) {
2     if(bytes(a).length != bytes(b).length) {
3         return false;
4     } else {
5         return keccak256(a) == keccak256(b);
6     }
7 }
```

Listing 4.15: String Equality Comparison pattern sample code

The function takes two strings as input parameters and returns true, if the strings are equal and false otherwise. In line 2 the strings are cast to bytes and their length is compared. In case of different lengths the function terminates and returns false. If the lengths match, the Keccak256 hashes of both parameters are calculated in line 5 and the result of their comparison is returned.

Gas Analysis

To quantify the potential reduction in required gas, a test has been conducted using the online solidity compiler Remix. Three different functions to check strings for equality have been implemented:

1. Check with the use of hashes.
2. Check by comparing each character; including length check.
3. Check with the use of hashes; including length check.

To account for different usage environments, a set of different input pairs has been used that covers short, medium and long strings, as well as matches and differences in early and late stages. The experimental code can be found on GitHub[50].

The results of the evaluation are shown in Table 4.2 and allow several conclusions:

- Checking with the help of hashes (Option 1 & 3) is more gas efficient than comparing characters, as soon as more than two characters would have to be compared. This is the case for matching strings with over two characters or pairs where the difference occurs only after the second position.
- In case of different lengths of the strings, methods that compare the strings before making any other tests (Option 2 & 3) are approximately 40% more efficient than options who do not do this check, regardless of the length of the strings.
- The additional gas usage when using a length check with the hash comparison is only around 3%, while it has the potential to save around 40% of gas every time the lengths do not match.

[50]String Equality Comparison gas example: `https://github.com/fravoll/solidity-patterns/blob/master/StringEqualityComparison/StringEqualityComparisonGasExample.sol`

Table 4.2: Execution cost for different forms of string comparison in gas.

Input A	Input B	Hash	Character + Length	Hash + Length
abcdefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz	1225	7062	1261
abcdefghijklmnopqrstuvwxyX	abcdefghijklmnopqrstuvwxyz	1225	7012	1261
Xbcdefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz	1225	912	1261
aXcdefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz	1225	1156	1261
acXefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz	1225	1400	1261
abcdefghijkl	abcdefghijklmnopqrstuvwxyz	1225	690	707
a	a	1225	962	1261
ab	ab	1225	1156	1261
abc	abc	1225	1450	1261

- The required amount of gas for the functions using hashes (Option 1 & 3) is very stable compared to the one comparing characters (Option 2), where the required gas grows linear with every needed iteration.

Consequences

The consequences of our proposed implementation of a string equality check with the use of hashes and a length comparison can be evaluated in regards to correctness and gas requirement. Correctness can be assumed to be ideal, since the chance of two strings having the same hash without being equal is negligible low [94]. Gas consumption is in most cases not optimal. We showed in the Gas Analysis section that in the case of two very short strings, Option 2 was slightly more efficient, while in the other cases Option 1 was cheaper. But combined, our implementation makes a good trade off between the other options and performs only slightly worse in some cases but significantly better in the others. Thus making it the best option in most of the scenarios, when no exact prediction about input parameters can be made. Another benefit is that the required gas is very stable and does not grow linear with the length of the string, as it does in Option 2, making it a scalable even for very long strings.

Known Uses

Usage of this pattern in a production environment could not be observed up until the point of writing.

4.4.2 Tight Variable Packing

Intent

Optimize gas consumption when storing or loading statically-sized variables.

Motivation

As with all patterns in this category the main goal of implementation is the reduction of gas requirement. This pattern in special is easily applied and does not change any contract logic. All that has to be done, is writing suitable state variables in the correct order. To reduce the amount of gas used for deploying a contract and later on calling his functions, we make use of the way the EVM allocates storage. Storage in Ethereum is a key-value store with keys and values of 32 bytes each [11]. When storage is allocated, all statically-sized variables (everything besides mappings and dynamically-sized arrays) are written down one after another, in the order of their declaration, starting at position 0 [2]. The most commonly used data types (e.g. `bytes32`, `uint`, `int`) take up exactly one 32 byte slot in storage. This pattern describes how to save gas by using smaller data types (e.g. `bytes16`, `uint32`) when possible, as the EVM can then pack them together in one single 32 byte slot and therefore use less storage. Gas is then saved because the

EVM can combine multiple reads or writes into one single operation. The underlying behavior is also referred to as "tight packing" and is unfortunately, until the time of writing, not automatically achieved by the optimizer.

Applicability

Use the Tight Variable Packing pattern when

- you want to reduce contract interaction costs.
- you are using more than one statically-sized state variable and can afford to use variables of smaller sizes.
- you are using a struct consisting of more than one variable and can afford to use variables of smaller sizes.
- you are using a statically-sized array and can afford to use a variable of a smaller size.

Participants & Collaborations

In general, the only participant in this pattern is the contract implementing it. All other entities interacting with said contract will not be influenced in any way, as the changes only affect how data gets stored.

Implementation

As hinted in the Applicability section, this pattern can be used for state variables, inside structs and for statically-sized arrays. The implementation of this pattern is quite straight forward and can be separated into two tasks:

1. Using the smallest possible data type that still guarantees the correct execution of the code. For example postal codes in Germany have at most 5 digits. Therefore, the data type `uint16`[51] would not suffice and we would use a variable of the type `uint24`[52] allowing us to store every possible postal code.
2. Grouping all data types that are supposed to go together into one 32 byte slot, and declare them one after another in the code. It is important to group data types together, as the EVM stores the variables one after another in the given order. This is only done for state variables and inside of structs. Arrays consist of only one data type, so there is no ordering necessary.

[51] `uint16` can hold numbers until $2^{16} - 1 = 65535$
[52] `uint24` can hold numbers until $2^{24} - 1 = 16777215$

It is possible to store as many variables into one storage slot, as long as the combined storage requirement is equal to or less than the size of one storage slot, which is 32 bytes. For example, one `bool` variable takes up one byte. A `uint8` is one byte as well, `uint16` is two bytes, `uint32` four bytes, and so on. The storage requirement of the `bytes` data type is easy to remember, since, for example, `bytes4` takes exactly four bytes. So theoretically 32 `uint8` variables can be stored in the same space as one `uint256` can. This only works if the variables are declared one after another in the code, because if one bigger data type has to be stored in between, a new slot in storage is used.

Sample Code

As an example we show how to use the pattern in the context of a struct.

```
1  contract StructPackingExample {
2
3      struct CheapStruct {
4          uint8 a;
5          uint8 b;
6          uint8 c;
7          uint8 d;
8          bytes1 e;
9          bytes1 f;
10         bytes1 g;
11         bytes1 h;
12     }
13
14     CheapStruct example;
15
16     function addCheapStruct() public {
17         CheapStruct memory someStruct = CheapStruct(1,2,3,4,"a","b","c","d");
18         example = someStruct;
19     }
20 }
```

Listing 4.16: Tight Variable Packing pattern sample code with struct

In line 3 we describe a struct object that makes use of the Tight Variable Packing pattern. The eight variables need one byte of storage each and are not interrupted by a bigger type, so they can be packed into one storage slot, where they use 8 of the available 32 bytes. That means we could add more variables into the same storage slot. In line 17 we first initialize a struct object in memory before we write it to storage in line 18.

Gas Analysis

To quantify the potential reduction in required gas, a test has been conducted using the online solidity compiler Remix. The sample code presented above is compared to a solution that stores the exact same input data but does not use the smallest possible data types, and orders the variables in a way that prevents the EVM to use tight packing. So instead of writing all eight variables into one slot, eight slots are used. The code of the experiment can be found on GitHub[53]. The results are shown in Table 4.3. It can be seen that the gas cost of contract creation is approximately 12% cheaper, when not using smaller data types. This can be explained because the EVM usually operates on 32 bytes at a time. It has to use additional operations in order to reduce the size of an element from its original to its reduced size, in our case from `bytes32` to `bytes1`, which costs extra gas. This cost pays off after saving one of our structs to storage. In our example we save 7 storage slots which amounts to saved gas of around 64%. This considerable amount of gas is not only saved once, but every time a new instance of this struct is stored.

Table 4.3: Transaction cost for tightly and not tightly packed structs in gas.

	Tightly Packed Struct	Struct without Tight Packing
Contract Creation	133172	116560
Saving Struct to Storage	57821	161636

Consequences

Consequences of the use of the Tight Variable Packing pattern have to be evaluated before implementing it blindly. The big benefit comes from the substantial amount of gas that can potentially be saved over the lifetime of a contract. But it is also possible to achieve the opposite, higher gas requirements, when not implementing it correctly. The positive effect on gas requirements only works for statically-sized storage variables. Function parameters or dynamically-sized arrays do not benefit from it. On the contrary, as seen in the contract creation costs in Table 4.3, it is even more costly for the EVM to reduce the size of a data type compared to leaving it in its initial state. Another issue may arise when reordering variables to optimize storage usage, which is decreased readability. Usually variables are declared in a logical order. Changing this order could make it harder to audit the code and confuse users as well as developers.

Known Uses

Implementation of this pattern is difficult to observe, because it is hard to differentiate, if variable types and ordering is chosen with storage packing in mind or because of different reasons. Up until writing no contract could be observed that seemed to have implemented this pattern completely deliberate. One noteworthy example is Roshambo[54], a rock-paper-scissors game that stores each game in a struct. Moves as well as tiebreakers are stored in `uint8` variables, which enable tight packing. But it looks

[53]Tight Variable Packing gas example: `https://github.com/fravoll/solidity-patterns/blob/master/TightVariablePacking/TightVariablePackingGasExample.sol`

[54]Roshambo: `https://etherscan.io/address/0xad01fab133e6b9a3308a68931f768ec86e1ad281#code`

like this design decision was made without tight packing in mind, as it could be further optimized.

Another example can be found in the Etherization contract[55], a DApp that provides a civilization like game on the Ethereum blockchain. In this contract every player is stored in a struct. This time no smaller data types are used, even it would be possible without breaking the logic of the game. By doing this, the gas requirement of storing a new player could be reduced significantly.

4.4.3 Memory Array Building

Intent

Aggregate and retrieve data from contract storage in a gas efficient way.

Motivation

Interacting with the storage of a contract on the blockchain is among the most expensive operations of the EVM [11]. Therefore, only necessary data should be stored and redundancy should be avoided if possible. This is in contrast to conventional software architecture, where storage is cheap and data is stored in a way that optimizes performance. While most of the times the only relevant cost of queries in those systems is time, in Ethereum even simple queries can cost a substantial amount of gas, which has a direct monetary value. One way to mitigate gas costs is declaring a variable public. This leads to the creation of a getter in the background allowing free access to the value of the variable [2]. But what if we want to aggregate data from several sources? This would require a lot of reading from storage and would therefore be particularly costly.

By using this pattern we are making use of the `view` function modifier in Solidity, which allows us to aggregate and read data from contract storage without any associated costs. Everytime a lookup is requested, an array is rebuilt in memory, instead of saving it to storage. This would be inefficient in conventional systems. In Solidity the proposed solution is more efficient because functions declared with the `view` modifier are not allowed to write to storage and therefore do not modify the state of the blockchain[56]. All data necessary for the execution of these functions can be fetched from the local node. Since the blockchain state stays the same, there is no need to broadcast a transaction to the network. No transaction means no consumed gas, making the call of a view function free, as long as it is called externally and not from another contract. In that case, a transaction would be necessary and gas would be consumed.

Applicability

Use the Memory Array Building pattern when

[55]Etherization: `https://etherscan.io/address/0x3f593a15eb60672687c32492b62ed3e10e149ec6#code`

[56]The Solidity documentation [2] gives an overview over what is considered modifying the state

- you want to retrieve aggregated data from storage.
- you want to avoid paying gas when retrieving data.
- your data has attributes that are subject to changes.

Participants & Collaborations

Participants in this pattern are the implementing contract itself as well as an entity requesting the stored data. To achieve a completely free request, the request has to be made externally, meaning not from another contract inside the network, as this would lead to the need for a gas intensive transaction.

Implementation

The implementation of this pattern can be divided into two parts. Part one covers the way the requested data is stored, whereas part two explains the actual aggregation and retrieval of the data:

1. To make data retrieval convenient it makes sense to chose a data structure that is easy to iterate over. In Solidity this is achieved by an array. In cases where aggregation is necessary, the data usually has more than one attribute. This characteristic can be implemented by a custom data type in the form of a struct. Combining these requirements, we end up with an array of structs, with the struct containing all attributes of an item. Another indispensable part is a mapping, which keeps track of the number of expected data entries for every possible aggregation instance. This mapping will come into play in part two.

2. The aggregation is then performed in a view function, so that no gas is consumed. A problem that makes the task a little more difficult, is the fact that Solidity does not allow an array of structs as a return value of a function yet [95]. We therefore propose a workaround that only returns the IDs of the desired items. It is then the task of the requesting entity to use these IDs to query the structs one by one. As the state is not changed by these additional operations, the queries are free as well.

   To gather the IDs of the desired items we first create an array to store them. Since we are not allowed to change the contract state in a view function we will create this array in memory. In Solidity it is not possible to create dynamic arrays in memory [2], so we can now make use of the mapping containing the number of expected entries from part one, and use it as the length for our array. The actual aggregation is done via a for-loop over all stored items. The IDs of all items that fit into the aggregation schema are saved into the memory array and returned after all items have been checked. Since all this computation is performed on the local node and not by every node on the network it is no problem to do such an otherwise expensive iteration over a dynamical array, since we cannot run out of gas.

Sample Code

In this sample, we show how a collection of items can be aggregated over its owners.

```
 1 contract MemoryArrayBuilding {
 2 
 3     struct Item {
 4         string name;
 5         string category;
 6         address owner;
 7         uint32 zipcode;
 8         uint32 price;
 9     }
10 
11     Item[] public items;
12 
13     mapping(address => uint) public ownerItemCount;
14 
15     function getItemIDsByOwner(address _owner) public view returns(uint[]) {
16         uint[] memory result = new uint[](ownerItemCount[_owner]);
17         uint counter = 0;
18 
19         for (uint i = 0; i < items.length; i++) {
20             if (items[i].owner == _owner) {
21                 result[counter] = i;
22                 counter++;
23             }
24         }
25         return result;
26     }
27 }
```

Listing 4.17: Memory Array Building pattern sample code

In line 3, an example struct is defined containing several attributes, including the owner over which we will aggregate later on. The array in line 11 contains all existing instances of items. In line 13, we store the amount of items every address holds, which is necessary to initialize the memory array in line 16. The function to retrieve all IDs of items belonging to a certain address in line 15 contains the `view` modifier. In the for-loop starting in line 19, we iterate over all items and check if their owner corresponds to the one we are aggregating over (line 20). If the owners match, we store the ID in our array. After all items have been checked, the array is returned. It is now possible for the requesting entity to query the items by their respective IDs without the need for a transaction, since the `items` array in line 11 is public.

Gas Analysis

The analysis of gas consumption in this pattern is fairly easy. Again, the Solidity online compiler Remix is used to compute the required gas. The code of the experiment can be found on GitHub[57]. In our experiment we use the setting presented in the Sample Code section and initialize it with ten items, of which two belong to the examined address. We then call the `getItemIDsByOwner(address _owner)` function twice: one time from an external account and once from another contract. One of the two times the function contains the `view` modifier and one time it does not. The results can be found in Table 4.4 and show how only a combination from an external call and the view function leads to a free query, while the other combinations cost gas like a regular function call would, because an actual transaction is broadcasted to the network.

Table 4.4: Transaction cost for item ID retrieval by a view and a regular function in gas.

	View Function	Regular Function
External Call	0	32623
Internal Call	32623	32623

Consequences

The most obvious consequence of applying the Memory Array Building pattern is the complete circumvention of transaction costs, a benefit that can save a substantial amount of money in case the function is used frequently. An alternative to the proposed way would be to store one array for every instance we would want to aggregate over (e.g. for every owner). This would lead to significant gas requirements, as soon as we wanted to change the owner of one item. We would have to remove the item from one array, add it to another array, shift every element in the first array to fill the empty space as well as reducing the length of the array. All of theses are expensive operations on contract storage. To change an attribute in the proposed solution, only the actual attribute and the mapping that keeps track of the length would have to be changed.

But the pattern does not only come with benefits. By implementing it, we increase complexity. It is unintuitive to store all items in one array compared to having separate arrays. Also the concept of doing aggregation on every single call instead of aggregating once and storing it that way might be confusing in the beginning.

Known Uses

An inplementation of this pattern can be found in the infamous CryptoKitties[58] contract. In line 651 we find a function called `tokensOfOwner(address _owner)` which returns the IDs of all Kitties that belong to a given address.

[57]Memory Array Building gas example: `https://github.com/fravoll/solidity-patterns/blob/master/MemoryArrayBuilding/MemoryArrayBuildingGasExample.sol`

[58]CryptoKitties contract: `https://etherscan.io/address/0x06012c8cf97bead5deae237070f9587f8e7a266d#code`

Another example is the now closed, Ethereum based slot machine Slotthereum[59]. In this contract, the pattern was used in a similar fashion as in our example, to retrieve the IDs of all games.

[59]Slotthereum contract: `https://etherscan.io/address/0xda8fe472e1beae12973fa48e9a1d9595f752fce0#code`

5 Evaluation of Patterns

This chapter deals with the evaluation of the programming patterns presented in the pattern catalog in chapter 4. A thorough evaluation, which mainly consists of validation and the elemination of errors, is of particular importance in this thesis, because the results include reusable pieces of codes. Even though the presented pieces of code are relatively short and seem clear at first sight, it is necessary to review each pattern throughout in order to exclude any possible attack vectors. This is a difficult task, considering there are no standardized testing procedures yet and testing tools are immature and scarce. None of the common smart contract analysis tools is able to detect all major security flaws [96]. Additionally, such analysis tools are designed for analyzing complete smart contracts. Most of the presented code samples, however, are incomplete contracts, as certain functionalities have been omitted for the sake of readability, or to keep the samples adaptable. These reasons eliminate the usage of analysis tools for pattern evaluation.

Our used evaluation approach consists of four different parts: evaluation via **literature**, evaluation via **testing**, evaluation via **community** and evaluation via **experts**. Each of these parts is featured in its own section hereinafter.

It should be further noted that even this throughout approach cannot guarantee that the provided patterns do not contain any undiscovered loopholes or other flaws. Every smart contract developer using any of the provided patterns, especially when adding or modifying any functionality, has the responsibility of thoroughly testing his code, in order to make sure it works as intended and does not contain any bugs.

5.1 Evaluation via Literature

Evaluation via literature applies to all patterns that are derived or inspired by practices that have been already published. For example, the State Machine pattern presented in subsection 4.1.2 was derived in modified form from the Solidity documentation. The modifications done to the patterns, in order to make them compatible with the remaining catalog, are usually only minor and should not introduce any security issues. Nevertheless, further testing is inevitable.

Patterns that have been derived or inspired by already existing practices have the benefit that they have most probably already been tested in some way. Usually, the author of the publication should have taken the task of making sure that the proposed technique is valid. Additionally, depending on the reach of the publication, a certain amount of readers participate in the evaluation process by raising concerns or pointing out possible mistakes, for example via comments. The target audience of such publications consists of interested and experienced smart contract developers with an existing background knowledge in that field. Which makes their comments a valuable tool for the validation process.

5.2 Evaluation via Testing

An evaluation via testing was conducted for each of the proposed patterns, particularly the code samples. The performed tests can be divided into two parts: **static** and **dynamic** analysis. While during static testing the code and the requirements are manually checked, dynamic testing includes the execution of code, in order to check its functional behavior [97].

- **Static testing**: For static testing, we conducted a static code review. This review included the manual inspection of written code, during and after implementation, for any syntax or logical errors. Additionally to the manual inspection, the online compiler Remix was used to identify compile-time errors and warnings.
- **Dynamic testing**: For dynamic testing, we conducted several manual unit tests for each pattern. These were mostly carried out by deploying the smart contract to the Ropsten test net[1] and consequently calling all possible functions with different inputs. Both, plausible and deliberately faulty inputs were used to check if the contracts were handling them appropriately.

For any discovered flaw, the cause was resolved and testing was repeated. This cycle continued until no more mistakes could be identified.

5.3 Evaluation via the Community

The ulterior motive behind evaluation with the help of the Ethereum developer community was to get feedback from a broad amount of knowledgeable and experienced developers. Therefore, the preliminary patterns were made publicly available on a web page via GitHub [98]. GitHub provides the possibility for every registered user to create *issues* with remarks or suggestions for improvements. Additionally, it is possible to submit self-created improvements or corrections via so called *pull requests*. We were hoping to be able to integrate the experience of the community via these methods and start a discourse about the patterns that would lead to their refinement.

A link to the web page containing the patterns, together with a short explanation of their background and a motivation for the participation via the creation of issues and pull requests, was published in selected forums and chat rooms. These selected

[1]The Ropsten test net is a blockchain like Ethereum, which is used as a testing environment before deploying smart contracts on the actual Ethereum blockchain, also called the main net. Ether for the Ropsten test net can be requested for free, avoiding a costly testing process on the main net.

channels are frequented by newcomers, dedicated developers, as well as the responsible persons behind Ethereum and Solidity. Amongst other places, the link was posted to the reddit channel /r/ethdev[2] (11,665 readers), a news aggregation site for development on Ethereum, and the Solidity Gitter[3] (5053 readers), a chat room for developers and users of Solidity.

The published patterns were well received by the community, gaining a lot of positive feedback (e.g. praising the useful information) and taking the first place of the trending Solidity projects on GitHub for over a week. However, the hoped-for discourse did not happen. Only two issues were created, of which the feedback was not related to the actual patterns. Publishing the preliminary results turned out to be a good way to spread the findings, but was not a suitable method for getting constructive feedback.

5.4 Evaluation via Experts

The evaluation via experts turned out to be more fruitful than via the community. We could convince Alexandros Papageorgiou, a blockchain specialist from the IT and service provider Inlecom, to be our expert and review the proposed patterns. The expert was able to voice a profound second opinion on our work by providing detailed annotations and suggestions for improvement. The feedback provided by the expert was thoroughly evaluated. Indisputable proposals were incorporated, while others were rejected. The rejected proposals included, amongst other things, minor changes that would have increased efficiency at the cost of readability. We decided to favor readability over efficiency, as long as the missed improvement was justifiable.

The revised document, including our reasoning for rejected changes, was sent back to the expert, who in return commented on the changes. This iterative approach is also called *shepherding* [99]. The unprejudiced and profound opinions of an independent expert were an important factor in evaluating and improving the quality of the patterns.

[2]Reddit channel /r/ethdev: `https://www.reddit.com/r/ethdev/`
[3]Solidity Gitter: `https://gitter.im/ethereum/solidity/`

6 Investigation of Pattern Usage

Besides the patterns presented in this work, several best practices for smart contract programming have been proposed since the introduction of Solidity. Some of these publications are referenced in chapter 3. From a research perspective it would be interesting to see if smart contract developers are actually making use of these reusable pieces of code and if they do, how long it takes for them to achieve widespread adoption. This knowledge could be beneficial in order to get an idea of how useful pattern catalogs, like the one presented in chapter 4 are, and which patterns are the most commonly used ones. To make a first step towards answering these questions, this chapter describes our investigation on pattern usage on the Ethereum blockchain. Particularly, we examine the spread of two contracts: the Ownable contract from OpenZeppelin [77], a concrete implementation of the Access Restriction pattern from subsection 4.2.1, and Oraclize, the leading oracle provider for the Oracle pattern, explained in subsection 4.1.3.

We start by explaining the approach that has been taken in order to investigate patterns on the blockchain. Afterwards we present our results regarding the two mentioned contracts, before we finish this chapter with a conclusion about our findings and possible further steps.

6.1 Investigation Approach

Smart contracts on the blockchain are stored in the form of byte code, which is hard to evaluate and cannot be fully reverted back into Solidity code. Block explorers, like etherscan.io[1], allow developers to provide matching source code to the published byte code. But only few contracts can be retrieved this way. To be able to conduct our research on the total quantity of published smart contracts, and not only the ones provided by etherscan, we make use of the way function identifiers work with the EVM. A function identifier is a short keyword that has to be provided with a transaction, when calling a function of a smart contract. It lets the EVM know, which function should be executed. It can be calculated by taking the first four bytes of the keccak256 hash of the function name in combination with the data types of the input parameters. For example the function `function helloWorld(uint a)public ..` has the identifier `5d3a9c29`, which is the result of `bytes4(keccak256("helloWorld(uint256)"))`. These function identifiers are also part of the contract byte code. This allows us to calculate the function identifiers of functions used in the patterns we want to examine and check each deployed contract

[1]Verified contracts on etherscan: `https://etherscan.io/contractsVerified`

for this hash. While hash collisions are possible, they are rather unlikely[2] and can be neglected in a study like this, which is only looking for trends. It is further possible, that some contracts contain functions with the exact name and combination of input parameters, without ever having heard of the pattern we are searching for. This fact has to be kept in mind, but the more unusual the function name is, the lower are the chances that someone implemented the function without being inspired by the pattern. The usage of function identifiers to draw conclusions about function names, however, is only possible for public or external functions, which excludes internal or private functions.

To get access to the total quantity of smart contract byte code in an efficient way, all contracts are stored in an *Elasticsearch* search engine. This is done by examining every single transaction from block 0 until block 5,501,900 (which was mined on 25.04.2018) on a local *geth* node, with the help of a *JavaScript* tool and the *web3.js* library. The tool checks each transaction, if it has the address 0 as an addressee, in which case it is creating a contract. The attached contract byte code is then stored in the search engine, together with the block number and its timestamp. This method only takes contracts created by transactions into account. Unfortunately, it is not able to fetch contracts that have been created by other contracts, as these are included in so called internal transactions. This limitation is accepted, in order to reduce the complexity of acquiring the data. The proposed method should still be able to deliver meaningful data, since it can be assumed that a contract creating other contracts uses the same code over and over again, which would exclude them at a later step anyways. Using this method we are able to gather the byte code of 1,600,447 smart contracts.

In order to avoid skewed data because of large stacks of identical contracts, we get rid of exact duplicates and consider only the first publication of each contract. This leaves us with 175,833 unique contracts.

Due to an attempt to congest the Ethereum network with massive contract creations in early October of 2016 [100] a significant spike in contract creations in one particular week can be observed. Because these contracts can be seen as an attack on the network, they are not regular smart contracts. We decided to exclude them from any further examinations. This is done, by eliminating smart contracts whose byte codes start with `0x6100` or `0x73`, which omitted 15,925 and 4495 contracts respectively in that particular week. After this cleanup 155,413 contracts are left for evaluation. Their distribution over time can be observed in Figure 6.1.

It can be seen how there has been a steady increase in contract creations since the beginning of 2017.

6.2 Ownable Investigation

The first implementation of a pattern we investigate is the Ownable contract by OpenZeppelin [77]. This contract implements the Access Restriction pattern, presented in subsection 4.2.1, by providing basic authorization control mechanisms. The creator of a

[2]A manual inspection of 50 smart contracts led to an accuracy of 100%.

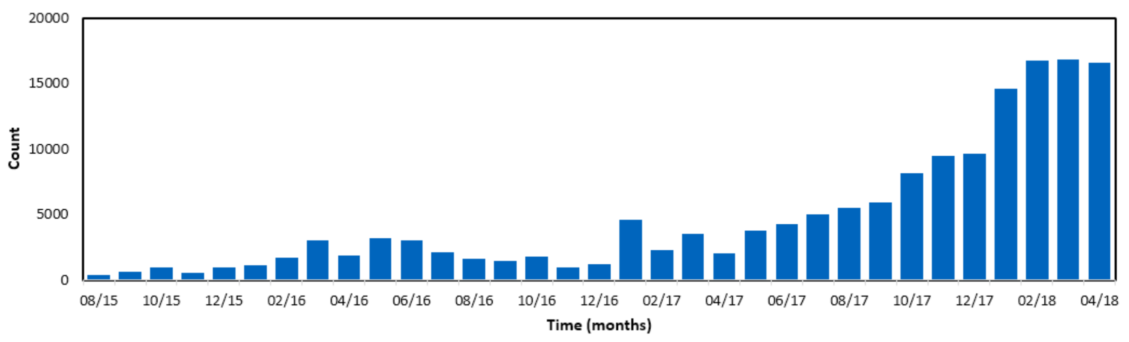

Figure 6.1: The monthly distribution of distinct smart contract creations after cleanup.

smart contract will be stored as the `owner` at contract creation time. With the help of a modifier, sensitive functions can be made available only to him. For example a function to change ownership to a new owner.

In order to identify contracts using this prefabricated code, we are searching for the two occurring functions in this contract: the function to transfer ownership and the getter function for the public address variable `owner`. This getter function is not explicitly stated in the code, but is automatically generated by the compiler, as for every public variable. The function identifier for `transferOwnership(address newOwner)` is `f2fde38b`, while the getter function for the owner has the identifier `8da5cb5b`. The search result for unique contracts containing these two functions can be found in Figure 6.2.

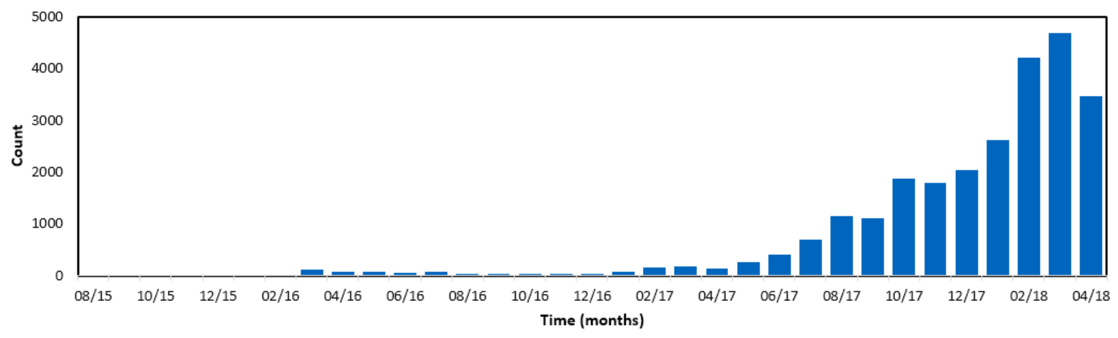

Figure 6.2: The monthly distribution of smart contracts implementing Ownable.

The graph shows the absolute numbers of contracts implementing the Ownable functionality. The distribution looks similar to Figure 6.1, only with less contracts in the beginning. The first version of this contract was published to GitHub in August 2016, however the function `transferOwnership(address newOwner)` was called `transfer(address newOwner)` at that time and was only changed in a commit from December 2016, to avoid overlaps with ERC20 tokens. This means that all occurrences before December 2016 have either used these two functions for their own implementations, or are internal tests that have been carried out before making the changes public. Either way, this figure alone only permits limited conclusions, as it lacks the reference to the total amount of contracts.

Therefore, we calculate the relative frequency of occurrences of the two functions and show them in Figure 6.3.

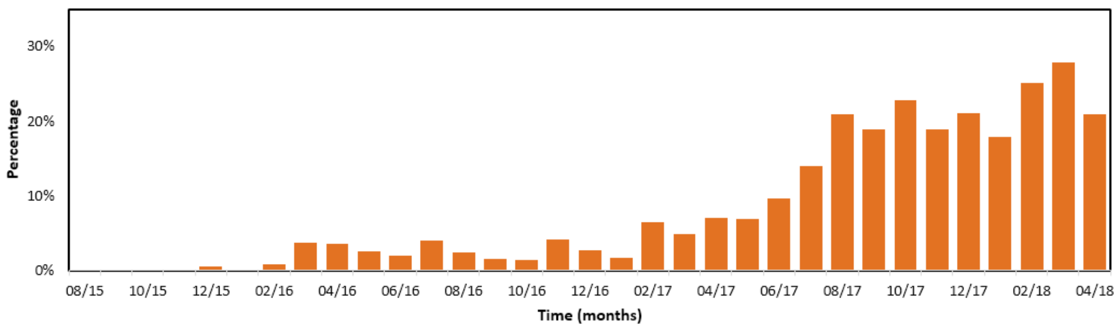

Figure 6.3: The monthly relative frequency of smart contracts implementing Ownable.

This figure clearly shows the significant increase in usage over time. Until mid 2017 under 10% of smart contracts implemented the Ownable functionality. From there on usage doubled to over 20% in August of 2017 with an increasing tendency in the following months. From evaluating this data it can be derived that over 20% of recently created smart contracts use the Ownable contract as a way to handle contract ownership. This is a considerable amount, taking into account that contracts, which implement the functionality but customize function names or add further input parameters, are not included in those numbers.

Another striking finding is that around 2.5% of unique contracts seem to have used the Ownable functionality before it was made public. In absolute values this is around 500 occurrences, so it could be linked to internal tests or coincidences.

6.3 Oraclize Investigation

We further examine the usage frequency of the service Oraclize. Oraclize is an external service that acts as a data carrier onto the blockchain [64]. It is a concrete implementation of the Oracle pattern from subsection 4.1.3. We are not interested in the contracts belonging to the service of Oraclize, but rather in the contracts using Oraclize as a service to get access to data from outside the blockchain. In order to search for contracts fitting this criterion, we make use of the requirement for using an oracle service: the implementation of a callback function. Every contract that wants to receive data from an oracle has to implement a callback function that can be called by the oracle to deliver the requested information.

The Oraclize service supports two different callback functions: the first function `function __callback(bytes32 myid, string result)` differs from the second function `function __callback(bytes32 myid, string result, bytes proof)` in only one parameter. Because the second function allows for a proof of the authenticity of the data to be passed as a third parameter, the two functions correspond to different identifiers. The first function, without the proof, matches the identifier `27dc297e`, while the second one has the string `38bbfa50` as its hash. Because both functions can be used to receive data from Oraclize, we count every smart contract containing either one, or both, of the two identifiers as using Oraclize as a service. The result for the search of these hashes on our database of unique contracts can be observed in Figure 6.4.

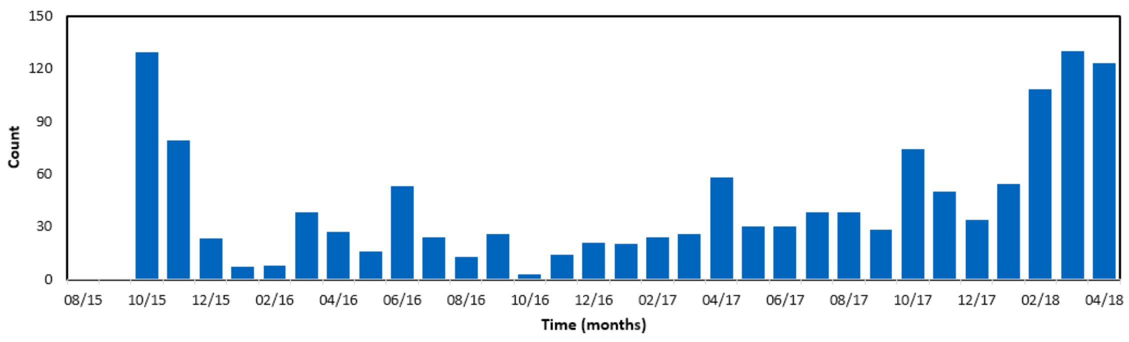

Figure 6.4: The monthly distribution of smart contracts using Oraclize as a service.

It has to be noted that the absolute numbers for this examination are several magnitudes lower than the ones from section 6.2. This could stem from the fact that using an oracle provider is associated with costs and is therefore only used in financially rewarding cases. When evaluating the chart, the comparatively high number of contracts using Oraclize in October and November of 2015 are particularly striking. After closer examination of these contracts on the blockchain, it could be determined that all these contracts have been created from the same address. It is therefore assumed that these contracts were used as tests by the developer of the service. This would match with the announcement date of the release of Oraclize shortly thereafter in November of 2015 [101]. In order to get a better idea of the actual distribution of the service, we once again calculated the relative frequency of occurrences, which are shown in Figure 6.5.

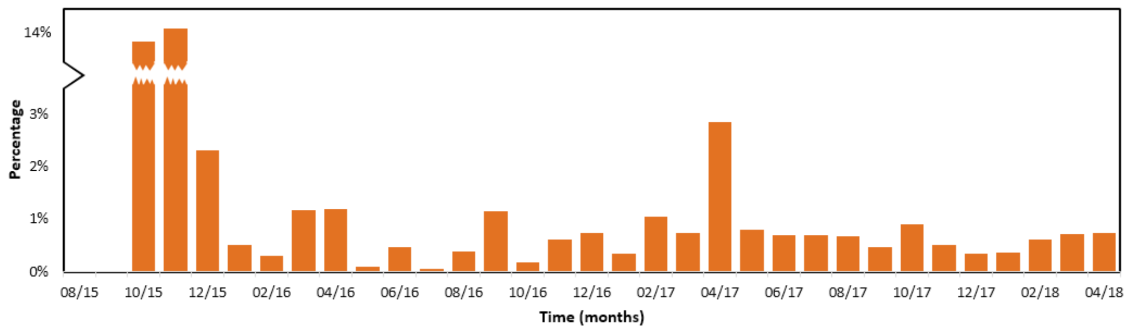

Figure 6.5: The monthly relative frequency of smart contracts using Oraclize as a service.

These results validate our assumption that a large percentage - almost 15% - of contracts used Oraclize as a service in the first two month of existence. After this initial spike, what we assume to be tests, usage dropped to around 1%. After some minor fluctuations, relative usage numbers settled and it can be concluded that at the time of writing, a little less than 1% of recently created contracts are using Oraclize as a data provider. This is a reasonable amount, considering the financial implications an external oracle brings with it and the necessary knowledge needed to implement such a solution.

6.4 Conclusion on Pattern Usage

In order to wrap up this investigation, it hast to be noted that the examination of only two cases does not allow any reliable conclusion for the entire smart contract ecosystem. However, it gives a first idea of possible usage percentages and is a starting point for further research that could be conducted on this topic. In the case of the Ownable contract in section 6.2, it has been shown that it took only two months to double the relative usage from under 10% to over 20%. This proves that it is possible for well acknowledged patterns and practices to be picked up by developers and achieve widespread adoption in a relatively short amount of time. Even though, it should be noted that the Ownable contract is probably one of the most used cases. It has further been shown that the obtained data has to be taken with care, as data can be skewed for several reasons, such as in the Oraclize example from section 6.3.

For future investigations in this area and work on this topic, contracts that were created by other contracts, which have not been included in this study, could be considered. This could be achieved by tracing the transactions to see if they lead to a contract creation after calling a contract, as it is done in [15]. The experimental setup presented in this chapter, can be used to check for occurrences of any possible function identifier in the matter of a second. Making it possible to investigate pattern distribution for each pattern containing publicly accessible methods and having somewhat special function names, which allow unambiguous identification. It could be the backbone of an online service that lets users get information on the usage of certain function names and provide graphic results, like the ones presented in this chapter.

7 Future Work and Conclusion

The aim of this chapter is to wrap up this thesis, by giving an outlook on possible future work, based on this research, in section 7.1 and summarizing the obtained results in section 7.2. Because possible extensions and future work on the topic of pattern usage investigation have already been discussed in section 6.4, the following section focuses on future work on the pattern catalog.

7.1 Future Work

Identification of further patterns - The pattern catalog presented in chapter 4 does not claim to be a complete compilation of all existing Solidity patterns. Rather, it should be seen as a starting point for a repository of gathering information, regarding best practices in smart contract development. There are probably countless methods of solving problems that could be researched and appended to this catalog. Furthermore, Solidity as a language is evolving. During the time of writing five alterations of Solidity[1] have been released, which introduced new functionality and eliminated bugs. With future updates, for example the unreleased version of 0.5.0, some of the proposed patterns could become obsolete. However, also new patterns could evolve, due to new possibilities in expressing functionality. These changes should be reflected in the pattern catalog.

Identification of new categories - As stated before, Solidity and its possibilities are evolving. With it, the whole ecosystem is changing. Smart contracts and DApps are getting more and more complex. In the future, new categories of pattern might be of interest, in order to account for new mechanisms and use cases. For example, economic patterns (see section 4.4) might not be of high relevance today, but with rising Ether prices, optimizing code will become a necessity. Similar categories that cannot be thought of at the current point, might play a big role in the future. To stay one step ahead, it should be thought of what developers will be in need of, if Ethereum evolves further in the direction it is headed to.

Expanding the scope of pattern application - Right now, the proposed pattern catalog contains idioms and design patterns. However, everything indicates that smart contracts will not be used on their own, but in connection with other software, like web front ends, in the future. To account for this development, it would be beneficial to also think about

[1]From version 0.4.19 to 0.4.24

the inclusion of architectural patterns that provide solutions for the apt integration of smart contracts into bigger applications.

Expanding the testing of existing patterns - Even though, the presented patterns have been evaluated thoroughly (see chapter 5), there is always a chance that bugs can be overseen. Having a vulnerability in a pattern that is supposed to be a reusable piece of code, is a bad thing. Even if the patterns are bug free at the moment, future updates to Solidity might break some functionality or introduce new bugs into already existing code. Therefore, testing should be continued. Tests could be conducted by using tools, which might yet have to be developed, gaining feedback by other experts on smart contract development and testing the patterns in more real-life scenarios.

7.2 Conclusion

The main questions answered by this thesis revolve around patterns in the smart contract programming language Solidity and the investigation of their usage. The research examined what challenges smart contract developers are currently facing. We answered the question, if there are patterns in Solidity to be identified and how they can be structured and classified. Furthermore, we had a look at methods for quantifying pattern usage and used that in order to draw conclusions about the willingness to use prefabricated solutions and the time it takes to adopt them.

After using grounded theory methods to gather information, we were able to identify and describe 14 different Solidity patterns. Each pattern is presented in an accessible way, including a motivation, its forces and a code sample. Additionally, four categories of patterns have been worked out, to which the patterns could be assigned. Further, we presented a method that enables the examination of pattern usage in smart contracts on the basis of byte code. The novelty about this approach is that no Solidity source code is needed, because the approach makes use of the hashes of function identifiers. We implemented this method and used it to investigate usage of two concrete applications of patterns: Ownable as an implementation of the Access Restriction pattern and the service Oraclize as an example for the Oracle pattern. Our research shows that Ownable is used in around 20% of currently deployed smart contracts that have not been deployed in the past. Oraclize is used as an oracle provider in a little less then 1% of contracts. For the case of Ownable, we showed that it is possible for adoption, to significantly increase in the matter of two months. Generally it could be shown that developers seem to be willing to use patterns in their contracts.

The result of this work is an extensible pattern catalog that can be helpful to beginners and experienced smart contract developers alike, improving their understanding of Solidity, as well as the security and functionality of their developed smart contracts. It is one step in the direction of more secure and reliable smart contracts, inevitable characteristics for the widespread adoption of a technology that could change the way humans and machines are going to interact in the future.

List of Figures

List of Tables

List of Listings

Bibliography

[1] S. Nakamoto. "Bitcoin: A peer-to-peer electronic cash system." In: (2008).

[2] EthereumFoundation. *Solidity Documentation*. `http://solidity.readthedocs.io/en/develop/`. Accessed: 2018-03-27.

[3] P. Daian. *Analysis of the DAO exploit*. `http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/`. Accessed: 2018-03-20. June 2016.

[4] B. Peterson. *The amount of ether frozen in digital wallets is worth $162 million – which is less than initially feared*. `https://www.businessinsider.com.au/ethereum-price-parity-hack-bug-fork-2017-11`. Accessed: 2018-04-05. Nov. 2017.

[5] S. Palladino. *The Parity Wallet Hack Explained*. `https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7`. Accessed: 2018-03-29. July 2017.

[6] K. Charmaz and L. L. Belgrave. "Grounded theory." In: *The Blackwell encyclopedia of sociology* (2007).

[7] A. M. Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies*. " O'Reilly Media, Inc.", 2014.

[8] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[9] V. Buterin. "A next-generation smart contract and decentralized application platform." In: 2014.

[10] B. Arvanaghi. *Reversing Ethereum Smart Contracts*. `https://arvanaghi.com/blog/reversing-ethereum-smart-contracts/`. Accessed: 2018-05-18. Mar. 2018.

[11] D. G. Wood. "Ethereum: a Secure Decentralised Generalised Transaction Ledger." In: 2014.

[12] N. Szabo. "Smart Contracts: Building Blocks for Digital Markets." In: *Extropy* 16 (1996).

[13] N. Szabo. *Smart Contracts*. `http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html`. Accessed: 2018-05-15. 1994.

[14] N. Szabo. "Formalizing and Securing Relationships on Public Networks." In: *First Monday* 2.9 (1997). ISSN: 13960466. DOI: `10.5210/fm.v2i9.548`.

[15] M. Fröwis and R. Böhme. "In Code We Trust? - Measuring the Control Flow Immutability of All Smart Contracts Deployed on Ethereum." In: *DPM/CBT@ESORICS*. 2017.

[16] K. Delmolino, M. Arnett, A. Kosba, A. Miller, and E. Shi. "Step by step towards creating a safe smart contract: Lessons and insights from a cryptocurrency lab." In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 79–94.

[17] D. Ellison. *An Introduction to LLL for Ethereum Smart Contract Development*. `https://media.consensys.net/an-introduction-to-lll-for-ethereum-smart-contract-development-e26e38ea6c23`. Accessed: 2018-05-11. May 2017.

[18] V. Buterin. *Comment on 'Does LLL have a future?'* `https://www.reddit.com/r/ethereum/comments/3tfg7i/does_lll_have_a_future/cx5ohet/`. Accessed: 2018-05-11. Nov. 2015.

[19] EthereumFoundation. *Serpent Wiki*. `https://github.com/ethereum/wiki/wiki/Serpent`. Accessed: 2018-05-11.

[20] EthereumFoundation. *Serpent GitHub repository*. `https://github.com/ethereum/serpent`. Accessed: 2018-05-11.

[21] ZeppelinSolutions. *Serpent Compiler Audit*. `https://blog.zeppelin.solutions/serpent-compiler-audit-3095d1257929`. Accessed: 2018-05-11. July 2017.

[22] V. Buterin. *PSA about Serpent's status*. `https://twitter.com/VitalikButerin/status/886400133667201024`. Accessed: 2018-05-11. July 2017.

[23] S. Tual. *Mutan FAQ*. `ttps://forum.ethereum.org/discussion/922/mutan-faq`. Accessed: 2018-05-11. Mar. 2015.

[24] G. Wood. *Solidity Publication*. `https://gist.githubusercontent.com/gavofyork/31b35cd2252a00d0d057/raw/16de06189d2175d2e31b300f1f8531e20c927635/solidity-original`. Accessed: 2018-05-11. Aug. 2014.

[25] S. Tual. *Solidity FAQ*. `https://forum.ethereum.org/discussion/1460/solidity-faq`. Accessed: 2018-05-11. Sept. 2014.

[26] EthereumFoundation. *Ethereum Documentation*. `http://ethdocs.org/en/latest/contracts-and-transactions/contracts.html`. Accessed: 2018-05-11.

[27] Hibryda. *Why Solidity isn't solid*. `https://medium.com/@Hibryda/why-solidity-isnt-solid-3341af77fc1c`. Accessed: 2018-05-11. June 2016.

[28] V. Buterin. *A few clarifications on Viper, Serpent and HLLs*. `https://www.reddit.com/r/ethereum/comments/5cpplr/a_few_clarifications_on_viper_serpent_and_hlls/`. Accessed: 2018-05-11. Nov. 2016.

[29] EthereumFoundation. *Vyper documentation*. `https://viper.readthedocs.io/en/latest/`. Accessed: 2018-03-15.

[30] pirapira. *Bamboo: a morphing smart contract language*. https://github.com/pirapira/bamboo. Accessed: 2018-05-11. Sept. 2016.

[31] Y. Hirai. *Bamboo compiler started producing EVM bytecode*. https://medium.com/@pirapira/bamboo-compiler-started-producing-evm-bytecode-6a55e4633de9. Accessed: 2018-05-11. June 2017.

[32] V. Buterin. *Thinking About Smart Contract Security*. https://blog.ethereum.org/2016/06/19/thinking-smart-contract-security/. Accessed: 2018-05-14. June 2016.

[33] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale." In: *CoRR* abs/1802.06038 (2018). arXiv: 1802.06038.

[34] N. Atzei, M. Bartoletti, and T. Cimoli. "A Survey of Attacks on Ethereum Smart Contracts (SoK)." In: *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. Ed. by M. Maffei and M. Ryan. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 164–186. ISBN: 978-3-662-54455-6. DOI: 10.1007/978-3-662-54455-6_8.

[35] D. Wong and M. Hemmel. *Decentralized Application Security Project Top 10 of 2018*. https://dasp.co/. Accessed: 2018-05-14. 2018.

[36] T. Cook, A. D. M. Latham, and J. H. Lee. "DappGuard : Active Monitoring and Defense for Solidity Smart Contracts." In: 2017.

[37] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. "Making Smart Contracts Smarter." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. Vienna, Austria: ACM, 2016, pp. 254–269. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978309.

[38] J. Jiao, S. Kan, S.-W. Lin, D. Sanan, Y. Liu, and J. Sun. "Executable Operational Semantics of Solidity." In: *arXiv preprint arXiv:1804.01295* (2018).

[39] J. Pfeffer. *Over 12,000 Ether Are Lost Forever Due to Typos*. https://media.consensys.net/over-12-000-ether-are-lost-forever-due-to-typos-f6ccc35432f8. Accessed: 2018-05-14. Mar. 2018.

[40] *Play Rock-Paper-Scissors for 1 ETH via Mist wallet*. https://www.reddit.com/r/ethtrader/comments/4fpn6o/play_rockpaperscissors_for_1_eth_via_mist_wallet/. Accessed: 2018-05-14. Apr. 2016.

[41] C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, Aug. 1977. ISBN: 0195019199.

[42] C. Alexander. *The Timeless Way of Building*. New York: Oxford University Press, 1979. ISBN: 0195024028.

[43] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[44] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996.

[45] G. Wagner. *The phenomenon of smart contract honeypots*. `https://medium.com/\spacefactor\@m{}gerhard.wagner/the-phenomena-of-smart-contract-honeypots-755c1f943f7b`. Accessed: 2018-03-29. Mar. 2018.

[46] M. Bartoletti and L. Pompianu. "An empirical analysis of smart contracts: platforms, applications, and design patterns." In: (Mar. 18, 2017). arXiv: `1703.06322v1 [cs.CR]`.

[47] P. Zhang, J. White, D. C. Schmidt, and G. Lenz. "Applying software patterns to address interoperability in blockchain-based healthcare apps." In: *arXiv preprint arXiv:1706.03700* (2017).

[48] A. Mavridou and A. Laszka. "Designing Secure Ethereum Smart Contracts: A Finite State Machine Based Approach." In: *arXiv preprint arXiv:1711.09327* (2017).

[49] M. Wöhrer and U. Zdun. "Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity." In: *1st International Workshop on Blockchain Oriented Software Engineering @ SANER 2018*. Mar. 2018.

[50] OpenZeppelin. *OpenZeppelin Solidity*. `https://github.com/OpenZeppelin/openzeppelin-solidity`. Accessed: 2018-05-14. Aug. 2016.

[51] cjgdev. *Smart-Contract Patterns*. `https://github.com/cjgdev/smart-contract-patterns`. Accessed: 2018-05-09. Aug. 2016.

[52] R. Simoes. *Solidity Patterns*. `https://github.com/robertsimoes/Solidity-Contract-Patterns`. Accessed: 2018-05-09. Dec. 2017.

[53] M. Wöhrer. *Smart-Contract Patterns*. `https://github.com/maxwoe/solidity_patterns`. Accessed: 2018-05-09. Mar. 2018.

[54] LoomNetwork. *Crypto Zombies*. `https://cryptozombies.io/`. Accessed: 2018-04-23. Jan. 2018.

[55] N. Dalal. *Learn X in Y minutes, where X = Solidity*. `https://learnxinyminutes.com/docs/solidity/`. Accessed: 2018-04-23. Nov. 2015.

[56] A. Beregszaszi. *Solidity Issue 1793 - Deprecate throw and suggest revert()/assert()/require() instead*. `https://github.com/ethereum/solidity/issues/1793`. Accessed: 2018-03-05. Mar. 2017.

[57] J. Maurelian. *The use of revert(), assert(), and require() in Solidity, and the new REVERT opcode in the EVM*. `https://media.consensys.net/when-to-use-revert-assert-and-require-in-solidity-61fb2c0e5a57`. Accessed: 2018-03-05. Sept. 2017.

[58] EthereumFoundation. *Ethereum - Block Protocol 2.0*. https://github.com/ethereum/wiki/blob/c02254611f218f43cbb07517ca8e5d00fd6d6d75/Block-Protocol-2.0.md. Accessed: 2018-03-08. Sept. 2015.

[59] EtherSport. *EtherSport Whitepaper - Innovative online sports lottery platform*. Nov. 2017.

[60] Ethorse. *Ethorse Whitepaper - The world's first Dapp to bet on the price of cryptocurrencies*. 2017.

[61] C. Mussenbrock. *Etherisc White Paper*. Nov. 2017.

[62] Orisi. *Orisi White Paper - The distributed oracles system for cryptocurrency contracts*. https://github.com/orisi/wiki/wiki/Orisi-White-Paper. Accessed: 2018-03-01. 2014.

[63] Oraclize. *Understanding oracles*. https://blog.oraclize.it/understanding-oracles-99055c9c9f7b. Accessed: 2018-03-01. Feb. 2016.

[64] Oraclize. *Oraclize Documentation*. https://docs.oraclize.it/. Accessed: 2018-01-17.

[65] P. Merriam. *How can a contract run itself at a later time?* https://ethereum.stackexchange.com/questions/42/how-can-a-contract-run-itself-at-a-later-time. Accessed: 2018-03-02. Jan. 2016.

[66] Y. N. Lee. *Data glitch: Google, Yahoo finance sites display incorrect stock market prices*. https://www.cnbc.com/2017/07/03/nasdaq-data-glitch-google-yahoo-display-incorrect-stock-market-prices.html. Accessed: 2018-03-02. July 2017.

[67] *IMPORTANT UPDATE: Mayweather/McGregor Smart Contract*. https://www.reddit.com/r/ethtrader/comments/6w5wcn/important_update_mayweathermcgregor_smart_contract/. Accessed: 2018-03-02. Aug. 2017.

[68] TLSNotaryGroup. *TLSnotary - a mechanism for independently audited https sessions*. Sept. 2014.

[69] E. Edgar. *Snopes meets Mechanical Turk: Announcing Reality Check - a crowd-sourced smart contract oracle*. https://medium.com/@edmundedgar/snopes-meets-mechanical-turk-announcing-reality-check-a-crowd-sourced-smart-contract-oracle-551d03468177. Accessed: 2018-03-02. Oct. 2017.

[70] Oraclize. *On decentralization of blockchain oracles*. https://blog.oraclize.it/on-decentralization-of-blockchain-oracles-94fb78598e79. Accessed: 2018-03-02. Jan. 2018.

[71] R. Kofler. *Random Generators for Ethereum contracts*. https://github.com/rolandkofler/ether-entrophy/. Accessed: 2018-01-17. July 2016.

[72] K. Edge. *Our thoughts on Ethereum, continued*. https://medium.com/@kpcb_edge/our-thoughts-on-ethereum-continued-3e7383c63779. Accessed: 2018-01-18. Dec. 2015.

[73] RANDAO. *RANDAO: A DAO working as RNG of Ethereum*. https://github.com/randao/randao. Accessed: 2018-03-12.

[74] J. Bonneau, J. Clark, and S. Goldfeder. *On Bitcoin as a public randomness source*. Cryptology ePrint Archive, Report 2015/1015. https://eprint.iacr.org/2015/1015. 2015.

[75] A. T. Griffith. *Is block.blockhash(block.number-1) okay?* https://medium.com/coinmonks/is-block-blockhash-block-number-1-okay-14a28e40cc4b. Accessed: 2018-04-19. Mar. 2018.

[76] E. Dimitrova. *Writing robust smart contracts in Solidity*. https://blog.colony.io/writing-more-robust-smart-contracts-99ad0a11e948. Accessed: 2018-03-15. Aug. 2016.

[77] OpenZeppelin. *Ownable.sol contract*. https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/ownership/Ownable.sol. Accessed: 2018-05-02. Apr. 2018.

[78] R. Hitchens. *Answer to Ethereum StackExchange question: Preparing for a throw, when forwarding ether to another contract?* https://ethereum.stackexchange.com/a/12233. Accessed: 2018-03-20. Feb. 2017.

[79] ConsenSys. *Ethereum Smart Contract Security Best Practices*. https://consensys.github.io/smart-contract-best-practices/. Accessed: 2018-03-20.

[80] C. Reitwießner. *Solidity Issue 610 - Add a "safe way to send ether" i.e. address.transfer*. https://github.com/ethereum/solidity/issues/610. Accessed: 2018-03-22. May 2016.

[81] *King of the Ether Throne - Post-Mortem Investigation*. http://www.kingoftheether.com/postmortem.html. Accessed: 2018-03-22. Feb. 2016.

[82] M. Araoz. *Onward with Ethereum Smart Contract Security*. https://blog.zeppelin.solutions/onward-with-ethereum-smart-contract-security-97a827e47702. Accessed: 2018-03-28. Aug. 2016.

[83] M. Inoue. *A SmartContract best practice: Push, Pull, or Give?* https://medium.com/@makoto_inoue/a-smartcontract-best-practice-push-pull-or-give-b2e8428e032a. Accessed: 2018-03-29. Nov. 2017.

[84] J. Tanner. *Summary of Ethereum Upgradeable Smart Contract R&D*. https://blog.indorse.io/ethereum-upgradeable-smart-contract-strategies-456350d0557c. Accessed: 2018-04-05. Mar. 2018.

[85] B. Marino and A. Juels. "Setting Standards for Altering and Undoing Smart Contracts." In: *Rule Technologies. Research, Tools, and Applications*. Ed. by J. J. Alferes, L. Bertossi, G. Governatori, P. Fodor, and D. Roman. Cham: Springer International Publishing, 2016, pp. 151–166. ISBN: 978-3-319-42019-6.

[86] N. Johnson. *Mad blockchain science: A 100% upgradeable contract*. https://www.reddit.com/r/ethereum/comments/4kt1zp/mad_blockchain_science_a_100_upgradeable_contract/. Accessed: 2018-04-05. May 2016.

[87] V. Buterin. *EIP 7: DELEGATECALL*. https://github.com/ethereum/EIPs/issues/23. Accessed: 2018-04-05. Nov. 2015.

[88] Daonomic. *Upgradeable Ethereum Smart Contracts*. https://medium.com/@daonomic/upgradeable-ethereum-smart-contracts-d036cb373d6. Accessed: 2018-04-06. Feb. 2018.

[89] E. Dimitrova. *Writing upgradable contracts in Solidity*. https://blog.colony.io/writing-upgradeable-contracts-in-solidity-6743f0eecc88. Accessed: 2018-04-09. June 2016.

[90] M. Calvanese. *Flexible Upgradability for Smart Contracts*. https://medium.com/level-k/flexible-upgradability-for-smart-contracts-9778d80d1638. Accessed: 2018-04-09. Mar. 2018.

[91] D. Rugendyke. *Upgradable Solidity Contract Design*. https://medium.com/rocket-pool/upgradable-solidity-contract-design-54789205276d. Accessed: 2018-04-10. Nov. 2017.

[92] Etherscan. *Ethereum Charts & Statistics*. https://etherscan.io/charts. Accessed: 2018-03-01.

[93] T. Chen, X. Li, X. Luo, and X. Zhang. "Under-optimized smart contracts devour your money." In: *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE. 2017, pp. 442–446.

[94] M. J. Dworkin. "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions." In: *Federal Inf. Process. Stds. (NIST FIPS) - 202* 202 (Aug. 2015).

[95] C. Reitwießner. *Solidity Issue 2948 - Return array of structs now support?* https://github.com/ethereum/solidity/issues/2948. Accessed: 2018-02-27. Sept. 2017.

[96] R. Fontein. "Comparison of static analysis tooling for smart contracts on the EVM." In: (2018).

[97] M. D. Ernst. "Static and dynamic analysis: Synergy and duality." In: *WODA 2003: ICSE Workshop on Dynamic Analysis*. 2003, pp. 24–27.

[98] F. Volland. *Solidity Patterns - A compilation of patterns and best practices for the smart contract programming language Solidity*. https://fravoll.github.io/solidity-patterns/. Accessed: 2018-05-08. Apr. 2018.

[99] T. Wellhausen and A. Fiesser. "How to write a pattern?: a rough guide for first-time pattern authors." In: *Proceedings of the 16th European Conference on Pattern Languages of Programs*. ACM. 2012, p. 5.

[100] M. H. Swende. *The Shanghai Attacks - From a technical perspective.* https://edcon.io/2017/ppt/one/Martin%20Holst%20Swende_The%20%27Shanghai%20%27Attacks_EDCON.pdf. Accessed: 2018-05-01. Feb. 2017.

[101] Oraclize. *Oraclize, the provably-honest oracle service, is finally here!* https://blog.oraclize.it/oraclize-the-provably-honest-oracle-service-is-finally-here-3ac48358deb8. Accessed: 2018-05-03. Nov. 2015.