Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity

Maximilian Wöhrer and Uwe Zdun
University of Vienna
Faculty of Computer Science
Währingerstraße 29, 1090 Vienna, Austria
Email: {maximilian.woehrer,uwe.zdun}@univie.ac.at

***Abstract*—Smart contracts that build up on blockchain technologies are receiving great attention in new business applications and the scientific community, because they allow untrusted parties to manifest contract terms in program code and thus eliminate the need for a trusted third party. The creation process of writing well performing and secure contracts in Ethereum, which is today's most prominent smart contract platform, is a difficult task. Research on this topic has only recently started in industry and science. Based on an analysis of collected data with Grounded Theory techniques, we have elaborated several common security patterns, which we describe in detail on the basis of Solidity, the dominating programming language for Ethereum. The presented patterns describe solutions to typical security issues and can be applied by Solidity developers to mitigate typical attack scenarios.**

I. Introduction

Ethereum is a major blockchain-based ecosystem that provides an environment to code and run smart contracts. Writing smart contracts in Solidity is so far a challenging undertaking. It involves the application of unconventional programming paradigms, due to the inherent characteristics of blockchain based program execution. Furthermore, bugs in deployed contracts can have serious consequences, because of the immediate coupling of contract code and financial values. Therefore, it is beneficial to have a solid foundation of established and proven design and code patterns that ease the process of writing functional and error free code.

With this paper we want to make the first steps in order to create an extensive pattern language. Our research aims to answer which code and design patterns commonly appear in Solidity coded smart contracts and the problems they intent to solve. In order to answer these questions we gathered data from different sources and applied Grounded Theory techniques to extract and identify the patterns.

This paper is structured as follows: First, we provide a short background to blockchain technology in Section II and the Ethereum platform in Section III. Then, we discuss some platform related security aspects in Section IV-C, before we present elaborated security patterns in Section V in detail. Finally, we discuss related work in Section VI, and draw a conclusion at the end in Section VII.

II. Background

A. Blockchains, Cryptocurrencies, and Smart Contracts

Blockchains are a digital technology that build on a combination of cryptography, networking, and incentive mechanisms to support the verification, execution and recording of transactions between different parties. In simple terms, blockchain systems can be seen as decentralized databases that offer very appealing properties. These include the immutability of stored transactions and the creation of trust between participants without a third party. That makes blockchains suitable as an open distributed ledger that can store transactions between parties in a verifiable and permanent way. One prominent application is the exchange of digital assets, so-called cryptocurrencies. Widely known cryptocurrencies are Bitcoin, Ethereum and Litecoin. They offer, beyond the transfer of digital assets, the execution of smart contracts. Smart contracts are computer programs that facilitate, verify, and enforce the negotiation and execution of legal contracts. They are executed through blockchain transactions, interact with crypto currencies, and have interfaces to handle input from contract participants. When run on the blockchain, a smart contract becomes an autonomous entity that automatically executes specific actions when certain conditions are met. Because smart contracts run on the blockchain, they run exactly as programmed, without any possibility of censorship, downtime, fraud or third party interference [1]. Today, the most-used smart contract platform in this regard is Ethereum.

III. Ethereum Platform

Ethereum is a public blockchain based distributed computing platform, that offers smart contract functionality. It provides a decentralised virtual machine as runtime environment to execute smart contracts, known as Ethereum Virtual Machine (EVM).

A. Ethereum Virtual Machine (EVM)

The EVM handles the computation and state of contracts and is build on a stack-based language with a predefined set of instructions (opcodes) and corresponding arguments [2]. So, in essence, a contract is simply a series of opcode statements, which are sequentially executed by the EVM. The EVM can be thought of as a global decentralized computer on which all smart contracts run. Although it behaves like one giant computer, it is rather a network of smaller discrete

machines in constant communication. All transactions, handling the execution of smart contracts, are local on each node of the network and processed in relative synchrony. Each node validates and groups the transactions sent from users into blocks, and tries to append them to the blockchain in order to collect an associated reward. This process is called mining and the participating nodes are called miners. To ensure a proper resource handling of the EVM, every instruction the EVM executes has a cost associated with it, measured in units of gas. Operations that require more computational resources cost more gas, than operations that require fewer computational resources. This ensures that the system is not jammed up by denial-of-service attacks, where users try to overwhelm the network with time-consuming computations. Therefore, the purpose of gas is twofold. It encourages developers to write quality applications by avoiding wasteful code, and ensures at the same time that miners, executing the requested operations, are compensated for their contributed resources. When it comes to paying for gas, a transaction fee is charged in small amounts of Ether, the built-in digital currency of the Ethereum network, and the token with which miners are rewarded for executing transactions and producing blocks. Ultimately, Ether is the fuel for operating the Ethereum platform.

B. Ethereum Smart Contracts

Smart contracts are applications which are deployed on the blockchain ledger and execute autonomously as part of transaction validation. To deploy a smart contract in Ethereum, a special creation transaction is executed, which introduces a contract to the blockchain. During this procedure the contract is assigned an unique address, in form of a 160-bit identifier, and its code is uploaded to the blockchain. Once successfully created, a smart contract consists of a contract address, a contract balance, predefined executable code, and a state. Different parties can then interact with a specific contract by sending contract-invoking transactions to a known contract address. These may trigger any number of actions as a result, such as reading and updating the contract state, interacting and executing other contracts, or transferring value to others. A contract-invoking transaction must include the execution fee and may also include a transfer of Ether from the caller to the contract. Additionally, it may also define input data for the invocation of a function. Once a transaction is accepted, all network participants execute the contract code, taking into account the current state of the blockchain and the transaction data as input. The network then agrees on the output and the next state of the contract by participating in the consensus protocol. Thus, on a conceptual level, Ethereum can be viewed as a transaction-based state machine, where its state is updated after every transaction.

C. Ethereum Programming Languages

Smart contracts in Ethereum are usually written in higher level languages and are then compiled to EVM bytecode. Such higher level languages are LLL (Low-level Lisp-like Language) [3], Serpent (a Python-like language) [4], Viper (a Python-like language) [5], and Solidity (a Javascript-like language) [6]. LLL and Serpent were developed in the early stages of the platform, while Viper is currently under development, and is intended to replace Serpent. The most prominent and widely adopted language is Solidity.

D. Solidity

Solidity is a high-level Turing-complete programming language with a JavaScript similar syntax, being statically typed, supporting inheritance and polymorphism, as well as libraries and complex user-defined types.

When using Solidity for contract development, contracts are structured similar to classes in object oriented programming languages. Contract code consists of variables and functions which read and modify these, like in traditional imperative programming.

```
pragma solidity ^0.4.17;
contract SimpleDeposit {
  mapping (address => uint) balances;

  event LogDepositMade(address from, uint amount);

  modifier minAmount(uint amount) {
    require(msg.value >= amount);
    _;
  }

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

  function deposit() public payable minAmount(1 ether)
      {
    balances[msg.sender] += msg.value;
    LogDepositMade(msg.sender, msg.value);
  }

  function getBalance() public view returns (uint
      balance) {
    return balances[msg.sender];
  }

  function withdraw(uint amount) public {
    if (balances[msg.sender] >= amount) {
      balances[msg.sender] -= amount;
      msg.sender.transfer(amount);
    }
  }
}
```

Listing 1. A simple contract where users can deposit some value and check their balance.

Listing 1 shows a simple contract written in Solidity in which users can deposit some value and check their balance. Before describing the code in more detail, it is helpful to give some insights about Solidity features like global variables, modifiers, and events.

Solidity defines special variables (`msg`, `block`, `tx`) that always exist in the global namespace and contain properties to access information about an invocation-transaction and the blockchain. For example, these variables allow the retrieval of the origin address, the amount of Ether, and the data sent alongside an invocation-transaction.

Another particular convenient feature in Solidity are so-called modifiers. Modifiers can be described as enclosed code units that enrich functions in order to modify their flow of code execution. This approach follows a condition-orientated

programming (COP) paradigm, with the main goal to remove conditional paths in function bodies. Modifiers can be used to easily change the behaviour of functions and are applied by specifying them in a whitespace-separated list after the function name. The new function body is the modifiers body where '_' is replaced by the original function body. A typical use case for modifiers is to check certain conditions prior to executing the function.

An additionally important and neat feature of Solidity are events. Events are dispatched signals that smart contracts can fire. User interfaces and applications can listen for those events on the blockchain without much cost and act accordingly. Other than that, events may also serve logging purposes. When called, they store their arguments in a transaction's log, a special data structure in the blockchain that maps all the way up to the block level. These logs are associated with the address of the contract and can be efficiently accessed from outside the blockchain.

Given this short feature description, we can now return and analyse the code example. First, the compiler version is defined (line 1), then the contract is defined in which a state variable is declared (line 3), followed by an event definition (line 5), a modifier definition (line 7), the constructor (line 12), and the actual contract functions (line 16 onwards). The state of the contract is stored in a mapping called `balances` (which stores an association between a users address and a balance). The special function `SimpleDeposit` is the constructor, which is run during the creation of the contract and cannot be called afterwards. It sets the balance of the individual creating the contract (`msg.sender`) to the amount of Ether sent along the contract creation transaction (`msg.value`). The remaining functions actually serve for interaction and are called by users and contracts alike. The `deposit()` function (line 16) manipulates the balances mapping by adding the amount sent along the transaction-invocation to the senders balance, while utilizing a modifier to preliminary ensure that at least 1 Ether is sent. The `withdraw()` function (line 25) manipulates the balances mapping by subtracting the requested amount to be withdrawn from the senders balance and the `getBalance()` function (line 21) returns the actual balance of the sender by querying the balances mapping.

In summary, this simple example shows the basic concepts of a smart contract coded in Solidity. Moreover, it illustrates the most powerful feature of smart contracts, which is the ability to manipulate a globally verifiable and universally consistent contract state (the balances mapping).

IV. Development Aspects

A. Limits of Blockchain Technology

First, it is important to state that not every application is predestined to be run on a blockchain. There are many applications that do not need a decentralized, immutable, append-only data log with transaction validation. Due to the inherent characteristics of blockchains, distributed ledger systems are not suitable for a variety of use cases. For example, computation-heavy applications are impractical to run on blockchains, because of the accumulated computation fees and the fact that many types of computations are impractical to execute on a stack-based virtual machine. Another limitation of blockchains is that they are not suitable for storing large amounts of data. This implicit limitation results in the extensive redundancy from the large number of network nodes, holding a full copy of the distributed ledger. Nonetheless, this can be overcome by not storing large data directly on the blockchain, but only a hash or other meta-data on the chain. In the context of data storage it is also important to realize that the data on the blockchain is visible to all network participants. This implies that keeping sensitive data confidential, requires the obfuscation of plaintext data by some means. A further limitation of blockchains is their performance. They are currently not suitable for applications which demand a high-frequency or low latency execution of transactions, because of the additional work owed to the cryptography, consensus, and redundancy apparatus of blockchain systems. Within these limits smart contracts should be used for applications that have something to gain from being distributed and publicly verifiable and enforceable. In general, most applications that handle the transfer or registration of resources in a traceable way are suitable, e.g. land register, provenance documentation, or electronic voting.

B. Coding Smart Contracts in Ethereum

Contract development on the Ethereum blockchain requires a different engineering approach than most web and mobile developers are familiar with. Unlike modern programming languages, which support a broad range of convenient data types for storage and manipulation, the developer is responsible for the internal organization and manipulation of data at a deeper level. This implies that the developer has to address details he may not be used to deal with. For example, a developer would have to implement a method to concat or lowercase strings, which are tasks developers usually do not have to think about in other languages. Furthermore, the Ethereum platform and Solidity are constantly evolving in a fast pace and the developer is confronted with an ongoing transformation of platform features and the security landscape, as new instructions are added, and bugs and security risks are discovered. Developers have to consider that code that is written today, will probably not compile in a few months, or will at least have to be refactored.

C. Smart Contract Security

An analysis of existing smart contracts by Bartoletti and Pompianu [7] shows that the Bitcoin and Ethereum platform mainly focus on financial contracts. In other words, most smart contract program code defines how assets (money) move. Therefore, it is crucial that contract execution is performed correctly. The direct handling of assets means that flaws are more likely to be security relevant and have greater direct financial consequences than bugs in typical applications. Incidents, like the value overflow incident in Bitcoin [8], or the DAO hack [9] in Ethereum, caused a hard fork of the blockchain

to nullify the malicious transactions. These incidents show that security issues have been used for fraudulent purposes ruthlessly in the past. A survey of possible attacks on Ethereum contracts was published by Atzei et al. [10] and lists 12 vulnerabilities that are assigned by context to Solidity, the EVM, and blockchain peculiarities itself. Many of these vulnerabilities can be addressed by following best practices for writing secure smart contracts, which are scattered throughout the Ethereum community [11, 12] and different Ethereum blogs. Most best practices mainly contain information about typical pitfalls to avoid and the description of favourable design and problem approaches. The latter being the focus of this paper in order to collate smart contract security design patterns.

V. Smart Contract Design Patterns

A software pattern describes an abstraction or conceptualization of a concrete, complex, and reoccurring problem that software designers have faced in the context of real software development projects and a successful solution they have implemented multiple times to resolve this problem [13].

So far, only few efforts have been made to collect and categorize patterns in a structured manner [14, 15]. This section gives an overview of typical security design patterns that are inherently frequent or practical in the context of smart contract coding. The presented patterns address typical problems and vulnerabilities related to smart contract execution. The patterns are based on multiple sources, such as review of Solidity development documentation, on studying Internet blogs and discussion forums about Ethereum, and the the examination of existing smart contracts. The source code of the presented patterns is available on github [16]. To illustrate the patterns in practice, Table I at the end of this section lists for each pattern an example contract with published source code deployed on the Ethereum mainnet.

A. Security Patterns

Security is a group of patterns that introduce safety measures to mitigate damage and assure a reliable contract execution.

1) Checks-Effects-Interaction:

CHECKS-EFFECTS-INTERACTION PATTERN
Problem When a contract calls another contract, it hands over control to that other contract. The called contract can then, in turn, re-enter the contract by which it was called and try to manipulate its state or hijack the control flow through malicious code.
Solution Follow a recommended functional code order, in which calls to external contracts are always the last step, to reduce the attack surface of a contract being manipulated by its own externally called contracts.

The Checks-Effects-Interaction pattern is fundamental for coding functions and describes how function code should be structured to avoid side effects and unwanted execution behaviour. It defines a certain order of actions: First, check all the preconditions, then make changes to the contract's state, and finally interact with other contracts. Hence its name is "Checks-Effects-Interactions Pattern". According to this principle, interactions with other contracts should be, whenever possible, the very last step in any function, as seen in Listing 2. The reason being, that as soon as a contract interacts with another contract, including a transfer of Ether, it hands over the control to that other contract. This allows the called contract to execute potentially harmful actions. For example, a so-called re-entrancy attack, where the called contract calls back the current contract, before the first invocation of the function containing the call, was finished. This can lead to an unwanted execution behaviour of functions, modifying the state variables to unexpected values or causing operations (e.g. sending of funds) to be performed multiple times. An example for a contract function, prone to the described attack scenario, is shown in Listing 3. The re-entrancy attack is especially harmful when using low level `address.call`, which forwards all remaining gas by default, giving the called contract more room for potentially malicious actions. Therefore, the use of low level `address.call` should be avoided whenever possible. For sending funds `address.send()` and `address.transfer()` should be preferred, these functions minimize the risk of re-entrancy through limited gas forwarding. While these methods still trigger code execution, the called contract is only given a stipend of 2,300 gas, which is currently only enough to log an event.

```
function auctionEnd() public {
  // 1. Checks
  require(now >= auctionEnd);
  require(!ended);
  // 2. Effects
  ended = true;
  // 3. Interaction
  beneficiary.transfer(highestBid);
}
```

Listing 2. Applying the Checks-Effects-Interaction pattern within a function.

```
mapping (address => uint) balances;

function withdrawBalance() public {
  uint amount = balances[msg.sender];
  require(msg.sender.call.value(amount)()); // caller's
      code is executed and can re-enter withdrawBalance
      again
  balances[msg.sender] = 0; // INSECURE - user's balance
      must be reset before the external call
}
```

Listing 3. An example of an insecure withdrawal function prone to a re-entrancy attack.

2) Emergency Stop (Circuit Breaker):

EMERGENCY STOP (CIRCUIT BREAKER) PATTERN
Problem Since a deployed contract is executed autonomously on the Ethereum network, there is no option to halt its execution in case of a major bug or security issue.
Solution Incorporate an emergency stop functionality into the contract that can be triggered by an authenticated party to disable sensitive functions.

Reliably working contracts may contain bugs that are yet unknown, until revealed by an adversary attack. One countermeasure and a quick response to such attacks are emergency stops or circuit breakers. They stop the execution of a contract or its parts when certain conditions are met. A recommended scenario would be, that once a bug is detected, all critical functions would be halted, leaving only the possibility to withdraw funds. A contract implementing the described strategy is shown in Listing 4. The ability to fire an emergency

stop could be either given to a certain party, or handled through the implementation of a rule set.

```
pragma solidity ^0.4.17;
import "../authorization/Ownership.sol";
contract EmergencyStop is Owned {
  bool public contractStopped = false;

  modifier haltInEmergency {
    if (!contractStopped) _;
  }

  modifier enableInEmergency {
    if (contractStopped) _;
  }

  function toggleContractStopped() public onlyOwner {
    contractStopped = !contractStopped;
  }

  function deposit() public payable haltInEmergency {
    // some code
  }

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

Listing 4. An emergency stop allows to disable or enable specific functions inside a contract in case of an emergency.

3) Speed Bump:

SPEED BUMP PATTERN
Problem The simultaneous execution of sensitive tasks by a huge number of parties can bring about the downfall of a contract.
Solution Prolong the completion of sensitive tasks to take steps against fraudulent activities.

Contract sensitive tasks are slowed down on purpose, so when malicious actions occur, the damage is restricted and more time to counteract is available. An analogous real world example would be a bank run, where a large number of customers withdraw their deposits simultaneously due to concerns about the bank's solvency. Banks typically counteract by delaying, stopping, or limiting the amount of withdrawals. An example contract implementing a withdrawal delay is shown in Listing 5.

```
pragma solidity ^0.4.17;
contract SpeedBump {
  struct Withdrawal {
    uint amount;
    uint requestedAt;
  }
  mapping (address => uint) private balances;
  mapping (address => Withdrawal) private withdrawals;
  uint constant WAIT_PERIOD = 7 days;

  function deposit() public payable {
    if(!(withdrawals[msg.sender].amount > 0))
      balances[msg.sender] += msg.value;
  }

  function requestWithdrawal() public {
    if (balances[msg.sender] > 0) {
      uint amountToWithdraw = balances[msg.sender];
      balances[msg.sender] = 0;
      withdrawals[msg.sender] = Withdrawal({
        amount: amountToWithdraw,
        requestedAt: now
      });
    }
  }

  function withdraw() public {
    if(withdrawals[msg.sender].amount > 0 && now >
        withdrawals[msg.sender].requestedAt + WAIT_PERIOD)
        {
      uint amount = withdrawals[msg.sender].amount;
      withdrawals[msg.sender].amount = 0;
      msg.sender.transfer(amount);
    }
  }
}
```

Listing 5. A contract that delays the withdrawal of funds deliberately.

4) Rate Limit:

RATE LIMIT PATTERN
Problem A request rush on a certain task is not desired and can hinder the correct operational performance of a contract.
Solution Regulate how often a task can be executed within a period of time.

A rate limit regulates how often a function can be called consecutively within a specified time interval. This approach may be used for different reasons. A usage scenario for smart contracts may be founded on operative considerations, in order to control the impact of (collective) user behaviour. As an example one might limit the withdrawal execution rate of a contract to prevent a rapid drainage of funds. Listing 6 exemplifies the application of this pattern.

```
pragma solidity ^0.4.17;
contract RateLimit {
  uint enabledAt = now;

  modifier enabledEvery(uint t) {
    if (now >= enabledAt) {
      enabledAt = now + t;
      _;
    }
  }

  function f() public enabledEvery(1 minutes) {
    // some code
  }
}
```

Listing 6. An example of a rate limit that avoids excessively repetitive function execution.

5) Mutex:

MUTEX PATTERN
Problem Re-entrancy attacks can manipulate the state of a contract and hijack the control flow.
Solution Utilize a mutex to hinder an external call from re-entering its caller function again.

A mutex (from mutual exclusion) is known as a synchronization mechanism in computer science to restrict concurrent access to a resource. After re-entrancy attack scenarios emerged, this pattern found its application in smart contracts to protect against recursive function calls from external contracts. An example contract is depicted below in Listing 7.

```
pragma solidity ^0.4.17;
contract Mutex {
  bool locked;

  modifier noReentrancy() {
    require(!locked);
    locked = true;
    _;
    locked = false;
  }
```

```
  // f is protected by a mutex, thus reentrant calls
  // from within msg.sender.call cannot call f again
  function f() noReentrancy public returns (uint) {
    require(msg.sender.call());
    return 1;
  }
}
```

Listing 7. The application of a mutex pattern to avoid re-entrancy.

6) Balance Limit:

BALANCE LIMIT PATTERN
Problem There is always a risk that a contract gets compromised due to bugs in the code or yet unknown security issues within the contract platform.
Solution Limit the maximum amount of funds at risk held within a contract.

It is generally a good idea to manage the amount of money at risk when coding smart contracts. This can be achieved by limiting the total balance held within a contract. The pattern monitors the contract balance and rejects payments sent along a function invocation after exceeding a predefined quota, as seen in Listing 8. It should be noted that this approach cannot prevent the admission of forcibly sent Ether, e.g. as beneficiary of a `selfdestruct(address)` call, or as recipient of a mining reward.

```
pragma solidity ^0.4.17;
contract LimitBalance {
  uint256 public limit;

  function LimitBalance(uint256 value) public {
    limit = value;
  }

  modifier limitedPayable() {
    require(this.balance <= limit);
    _;
  }

  function deposit() public payable limitedPayable {
    // some code
  }
}
```

Listing 8. A contract limiting the total balance acquirable through payable function invocation.

TABLE I
PATTERN USAGE EXAMPLES IN PUBLISHED SOURCE CODE CONTRACTS ON THE ETHEREUM MAINNET.

Category	Pattern	Example Contract
Security	Checks-Effects-Interaction	CryptoKitties
	Emergency Stop	Augur/REP
	Speed Bump	TheDAO
	Rate Limit	etherep
	Mutex	Ventana Token
	Balance Limit	CATToken

VI. Related Work

According to Alharby and van Moorsel [17] current research on smart contracts is mainly focused on identifying and tackling smart contract issues and can be divided into four categories, namely coding, security, privacy and performance issues. The technology behind writing smart contracts in Ethereum is still in its infancy stage, therefore coding and security are among the most discussed issues. Unfortunately, a lot of research and practical knowledge is scattered throughout blog articles and grey literature, therefore information is often not very structured. Only relatively few papers focus on software patterns in blockchain technology respectively on design patterns in the Solidity language for the Ethereum ecosystem. Bartoletti and Pompianu [7] conducted an empirical analysis of Solidity smart contracts and identified a list of nine common design patterns that are shared by studied contracts. These patterns broadly summarize the most frequent solutions to handle common usage scenarios. A paper by Zhang et al. [18] describes how the application of familiar software patterns can help to resolve design specific challenges. In particular, commonly known design patterns such as the Abstract Factory, Flyweight, Proxy, and Publisher-Subscriber pattern are applied to implement a blockchain-based healthcare application. The above mentioned papers do not contain security related design patterns, but show that design patterns are an interesting topic in smart contract coding.

VII. Conclusion

We have given a brief introduction to Ethereum and Solidity and outlined six design patterns that address security issues when coding smart contracts in Solidity. In general, the main problem that these patterns solve is the lack of execution control once a contract has been deployed, resulting form the distributed execution environment provided by Ethereum. This one-of-a-kind characteristic of Ethereum allows programs on the blockchain to be executed autonomously, but also has drawbacks. These drawbacks appear in different forms, either as harmful callbacks, adverse circumstances on how and when functions are executed, or uncontrollably high financial risks at stake. By applying the presented patterns, developers can address these security problems and mitigate typical attack scenarios.

In future work, we plan to extend the already collated patterns to create a structured and informative design pattern language for Solidity, that can be used as guidance for developers or find its application in automatic code generating frameworks.

References

[1] Ethereum project. [Online]. Available: https://www.ethereum.org/

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum Project Yellow Paper*, vol. 151, 2014.

[3] Lll poc 6. [Online]. Available: https://github.com/ethereum/cpp-ethereum/wiki/LLL-PoC-6/7a575cf91c4572734a83f95e970e9e7ed64849ce

[4] Serpent. [Online]. Available: https://github.com/ethereum/wiki/wiki/Serpent

[5] ethereum/viper: New experimental programming language. [Online]. Available: https://github.com/ethereum/viper

[6] Solidity — solidity 0.4.18 documentation. [Online]. Available: https://media.readthedocs.org/pdf/solidity/develop/solidity.pdf

[7] M. Bartoletti and L. Pompianu, "An empirical analysis of smart contracts: platforms, applications, and design patterns," *arXiv preprint arXiv:1703.06322*, 2017.

[8] Value overflow incident - bitcoin wiki. [Online]. Available: https://en.bitcoin.it/wiki/Value_overflow_incident

[9] P. Daian, "Analysis of the dao exploit," 2016, [Online; accessed 6-September-2017]. [Online]. Available: http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[10] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International Conference on Principles of Security and Trust*. Springer, 2017, pp. 164–186.

[11] Security considerations — solidity 0.4.18 documentation. [Online]. Available: http://solidity.readthedocs.io/en/develop/security-considerations.html

[12] "Ethereum contract security techniques and tips," 2017, [Online; accessed 6-September-2017]. [Online]. Available: https://github.com/ConsenSys/smart-contract-best-practices

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design patterns: elements of," 1994.

[14] J. Bontje. (2015) Dapp design patterns. [Online]. Available: https://www.slideshare.net/mids106/dapp-design-patterns

[15] cjgdev. (2016) Smart-contract patterns written in solidity, collated for community good. [Online]. Available: https://github.com/cjgdev/smart-contract-patterns

[16] "maxwoe/solidity_patterns." [Online]. Available: https://github.com/maxwoe/solidity_patterns

[17] M. Alharby and A. van Moorsel, "Blockchain-based smart contracts: A systematic mapping study," *arXiv preprint arXiv:1710.06372*, 2017.

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