Security Vulnerabilities in Ethereum Smart Contracts

Alexander Mense[†]
Department for Computer Science
University of Applied Sciences
Technikum Wien
Vienna, Austria
mense@technikum-wien.at

Markus Flatscher
Department for Computer Science
University of Applied Sciences
Technikum Wien
Vienna, Austria
flatscher@technikum-wien.at

ABSTRACT

Smart contracts (SC) are one of the most appealing features of blockchain technologies facilitating, executing, and enforcing predefined terms of coded contracts without intermediaries. The steady adoption of smart contracts on the Ethereum blockchain has led to tens of thousands of contracts holding millions of dollars in digital currencies and small mistakes during the development of SC on immutable blockchains have already caused substantial losses and involve the danger for future incidents. Hence, today the secure development of smart contracts is an important topic and several attacks and incidents related to vulnerable smart contracts could have been avoided. To foster a secure development process of SC this paper summarizes known vulnerabilities in smart contracts found by literature research and analysis. It compares currently available code analysis tools for their capabilities to identify and detect vulnerabilities in smart contracts based on a taxonomy for vulnerabilities. Finally, based on the TheDOA attack the paper shows an example for the adoption of best practices to avoid severe vulnerabilities in smart contracts.

CCS CONCEPTS

• Security and privacy → Distributed systems security; Software security engineering;

KEYWORDS

Blockchain, Smart Contracts, Security

iiWAS '18, November 19–21, 2018, Yogyakarta, Indonesia

ACM ISBN 978-1-4503-6479-9/18/11…$15.00
https://doi.org/10.1145/3282373.3282419

1 Introduction

A blockchain is a distributed system with a redundant database where addresses (the public half of a cryptographic key pair) are mapped to quantities of virtual money (cryptocurrency) like in a ledger. Everyone can connect to a blockchain's public peer-to-peer network, effectively sharing the same ledger. In combination with cryptography, blockchain can securely host applications, enable the transfer of digital assets (such as digital currencies), and the storage of data in a decentralized way. Under the pseudonym Satoshi Nakamoto, blockchain technology was introduced in 2008 in a paper called *Bitcoin: A Peer-to-Peer Electronic Cash System* [1] and was first implemented by the crypto currency Bitcoin the year after. While Bitcoin itself has got a lot of media attention over the years as the top performing currency in 2015, the best performing commodity in 2016, with more than 300.000 daily transactions as of May 2017 [2], the underlying technology can also be used for several applications beyond crypto currencies. Since transactions can be processed without any bank or intermediary, the application in financial services is the most evident example. Other promising areas include the Internet of Things (IoT), public services, reputation systems, security services, and smart contracts [3].

In 2015, Ethereum, a blockchain with a built-in Turing-complete[1] programming language, opened up new potential in the area of smart contracts [4]. As of the beginning of 2018 Ethereum is the second most popular blockchain platform based on the current market capitalization [5]. Besides acting as a crypto-currency (called Ether) similar to Bitcoin, the support of a Turing-complete programming language allows anyone to write advanced smart contracts and decentralized applications [4].

1.1 Smart Contracts

The concept of smart contracts was first proposed by Nick Szabo around 1997 [7]. A smart contract is a piece of executable code that runs on the blockchain to autonomously facilitate, execute, and enforce the pre-defined terms of an agreement without the involvement of a trusted third party. This results in low transaction costs, but also carries several security issues, such as

[1] A system is called Turing-complete if it can perform any logical step of the computational functions a universal Turing machine can.

the inherent immutability of blockchain when combined with a smart contract, i.e. the inability to change the contract's code once it has been deployed on the blockchain [6].

Usually smart contracts are termed as "software applications" to make the term more user-friendly and understandable, but actually they are more akin to classes in object-oriented programming than to software applications [8]. In [9], smart contracts are referred to as self-autonomous and self-verifying agents, composed of fields and functions, which are stored on the blockchain. Once deployed on the blockchain, smart contracts receive a unique identifier (contract account address) to differentiate them from users (with externally-owned accounts) who can interact with them [4]. Ethereum smart contracts are implemented by a low-level stack-based byte code, known as *"Ethereum virtual machine code"* or colloquially as *"EVM code"*. But usually they are developed in high-level, JavaScript-like language such as Solidity. Since Ethereum is classified as a public blockchain, it follows that the byte code of every smart contract is publicly available and every node in the network can inspect its code. Therefore, their behavior is predictable [4]. Smart contracts possess the functionality to hold a state, exchange digital assets, take input, store data, obtain information from external services, and express business logic. After deployment to the blockchain, the function of a smart contract is triggered by either messages or transactions sent to the contracts unique address [10]. Furthermore, a smart contract is considered deterministic, i.e. the same input always produces the same output. Non-deterministic contracts have the potential to break a blockchain. As every node in the network executes a non-deterministic contract, it returns different results, preventing consensus [11].

2 Methods

As a smart contract cannot be changed any more once it is deployed on the blockchain, security plays a major role in the developing such applications. Therefore, it is important for developers to know existing vulnerabilities, use best practices for secure coding throughout the development process and use tool support wherever possible to avoid erroneous implementations.

In a first step, based on a taxonomy for vulnerabilities resulting from literature research the paper describes categories of vulnerabilities known to impair the security of Ethereum smart contracts and discusses the different factors that cause them as well as how they can be exploited.

In a second step, results of a comparison of currently available code analysis tools for smart contracts and their detection capabilities for these vulnerabilities are presented.

Finally, based on the example of the TheDAO-attack coding examples for the development of secure Ethereum smart contracts and the avoidance of vulnerabilities are demonstrated.

3 Results

As literature analysis has revealed, the taxonomy proposed by Atzei et al. [12], is referenced several times by other researchers [13]–[15] and modified in [2] by appending the technical root causes for the vulnerabilities and in [16] by appending severity levels. They group vulnerabilities based on the architectural level where the vulnerability occurs: the high level programming language (i.e. Solidity), the EVM bytecode, or blockchain general characteristics. Solidity-level vulnerabilities apply to other high-level programming languages in Ethereum as well [12].

The taxonomy in Figure 1 is based on a taxonomy provided in [12] and was refined and extended by Dika [16]. Dika [16] argues that this taxonomy applies to newly discovered vulnerabilities as much as existing ones, since all vulnerabilities falls into one of the defined levels. The taxonomy includes high-severity vulnerabilities (assigned the value "3" in Figure 1), non-critical vulnerabilities (value "2") and warnings (value "1"). It has to be noticed that when used in critical components of smart contracts, all levels can lead to major security issues [16].

	Vulnerability	Severity
Solidity	Gas costly patterns	1-2
	Call to the unknown	3
	Gasless send	3
	Mishandled exceptions	3
	Type casts	2
	Re-entrancy	3
	Unchecked math (Integer over- / underflow)	1-2
	Exposed functions or secrets	2-3
	'tx.origin' usage	3
	'blockhash' usage	2-3
	Denial of Service (DoS)	3
	'send' instead of 'transfer'	1-2
	Style violation	1
	Redundant fallback function	1
EVM	Immutable bugs / mistakes	3
	Ether lost in transfer	3
Blockchain	Unpredictable state (dynamic libraries)	2
	Generating randomness	2-3
	Timestamp dependency	1-3
	Lack of transactional privacy	1-3
	Transaction-ordering dependence	2-3
	Untrustworthy data feeds (oracles)	3

Figure 1: Taxonomy of vulnerabilities in Ethereum smart contracts based on [12][16]

3.1 Vulnerabilities

This section gives brief explanations of known security vulnerabilities (see Figure 1). Primarily, the focus of this section is on severe vulnerabilities and vulnerabilities that are not self-explanatory.

External calls can introduce several risks or errors, as external contracts can execute malicious code. In general external calls or "calls to the unknown" should by avoided. In case interaction with external contracts is necessary, precautions such as marking untrusted contracts, avoiding state changes after external calls, and handling errors in external calls must be taken [17].

Gasless send refers to the fee model in Ethereum, requiring users who initiate a transaction to pay this fee in gas costs. This also includes costs for dependent calls. This means that users or contracts initiating calls to other contracts must pay for their

execution. A call made with insufficient gas aborts and throws an out-of-gas exception and, if not handled, all provided gas for the execution will be spent, i.e. there is no reimbursing of the spent gas [18].

Mishandled exceptions, occurs in the callee contract if an exception is raised (e.g., not enough gas). The contract then terminates, reverts its state, and returns false. Depending on how the call is made (send instruction or direct call of contract's function), the exception thrown by the callee may get propagated to the caller or not. Out of 19.366 Ethereum smart contracts 27.9% had mishandled exceptions, as of May 5, 2016 [19].

Re-entrancy was exploited in the TheDAO attack—the biggest attack on Ethereum smart contracts [12], [19]. The consequences this attack had on the Ethereum blockchain were unprecedented. The price of ether dropped from $20 to $13. TheDAO attack is also the reason why the Ethereum blockchain was split in two—Ethereum and Ethereum Classic—by a hard fork [20]. The vulnerability can be explained as follows: *"Any interaction from a contract (A) with another contract (B) and any transfer of Ether hands over control to that contract (B). This makes it possible for B to call back into A before this interaction is completed."* [21]

In doing so, contract (B) can retrieve refunds multiple times until the balance of contract (A) is emptied out. To avoid re-entrancy, it is recommended to use the checks-effectsinteractions pattern [21] (see subsection III-D).

Tx.origin is a global variable that returns the address of the sender of the message by retracing the full call chain, i.e. the origin of the call instead of the address used for the current call. It must not be used for the purpose of authorisation. Tx.origin would set the attacker address as contract owner, giving the attacker full access to the funds held by the affected contract, making it possible to instantly drain all the funds [21]. It is also worth mentioning that the interoperability between contracts is limited, because a contract cannot be tx.origin [22].

Timestamp dependency occurs in contracts using the block timestamp as a condition to trigger and execute a transaction. As the block timestamp is dependent on the local time of the miner who created the block, a dishonest miner could manipulate the block timestamp in his favour [6], [12]. A general rule of thumb is that if a contract function can tolerate variations by thirty seconds and still maintain integrity, block timestamps are safe to use [22]. In [23], it is stated that no precision of better than fifteen minutes can be expected from block timestamps.

Blockhash usage poses a similar issue as the use of the block timestamps. It is recommended, that crucial smart contract components refrain from using blockhash. As with timestamp dependency, the blockhash is susceptible to manipulation by miners [12].

Immutable bugs references one of the core concepts of blockchain: immutability, which applies to smart contracts as well. Once deployed to the blockchain, a contract can no longer be altered, which creates trust in the sense that if the contract implements the functionality as intended by developers, its runtime behaviour (ensured by the consensus protocol) will be the expected one. However, if a deployed contract contains a bug, patching the contract is impossible without building in ways during development to either alter or outright terminate vulnerable contracts [12]. Reconciling the immutability of the blockchain with Ethereum's philosophy is debatable though, as their own slogan reads: *"Ethereum is a decentralized platform that runs smart contracts: applications that run exactly as programmed without any possibility of downtime, censorship, fraud or third-party interference."* [24]

Ether lost in transfer happens when ether is sent to a recipient address that is an orphan, i.e. has no association with any contract or user. If ether is sent to an orphan address, it cannot be recovered. As there is no way to check whether or not an address is an orphan address, developers must manually ensure the correctness of recipient addresses [12].

Transaction-ordering dependency occurs when two dependent transactions invoke the same contract and are part of the same block. In such a scenario, there is a discrepancy between the contract state, which a caller wishes to invoke at, and the actual state when the execution happens. As the order of transaction execution relies on miners, an adversary can launch an attack if the correct transaction order is upended [19]. For example, when a user submits a solution (transaction t0) to a smart contract offering a reward for solving a computational problem and, at the same time, the owner reduces or removes the reward via a separate transaction (t1), it depends on how the miner orders the transactions when mining the next block. If t0 is processed first, the user receives the original reward. If t1 is processed first, the user receives a smaller or no reward [6].

Untrustworthy data feeds, or oracles, occur in smart contracts that require information from services outside the blockchain [6]. Data feeds are smart contracts on the blockchain that serve requests by other contracts to source data from outside the blockchain, such as from websites, but guarantee no certainty as to the trustworthiness of the information provided by external services [18].

3.2 Code Analysis Tools

Tool support can significantly enhance the development process for secure smart contracts. There are several tools with the ability to detect vulnerabilities in smart contracts during development and a first comparison of tools based on eight chosen entries from the presented taxonomy has been described in [16]. As part of this work a further complete comparison of the following tools has been conducted: Oyente [19], Securify [25], Remix [26], SmartCheck [27], F* [28], Mythril [29], and Gasper [15]. Their detection capabilities were matched against all of the vulnerabilities in the taxonomy presented in Figure 1.

Figure 2 shows the results of the detailed analysis of the vulnerability detection capabilities of the security tools for smart contracts, which are applicable during, but not necessarily limited to, pre-deployment on the blockchain.

The results of this paper indicate that re-entrancy ranks highest among the listed vulnerabilities (detectable by six out of seven tools), closely followed by mishandled exceptions and tx.origin

usage. Gas-costly patterns and timestamp dependency rank third, followed by exposed functions, blockhash usage, and transaction-ordering dependency. Less than two of the examined tools check the remaining vulnerabilities. When compared with Figure 1 the focus on severe vulnerabilities becomes obvious.

	Vulnerability	Oyente	Securify	Remix	SmartCheck	F*	Mythril	Gasper	Sum
Solidity	Gas costly patterns	✓	×	✓	✓	×	×	✓	4
	Call to the unknown	×	×	×	×	×	✓	×	1
	Gasless send	×	×	×	×	×	×	×	0
	Mishandled exceptions	✓	✓	✓	✓	✓	×	×	5
	Type casts	×	×	×	×	×	✓	×	1
	Re-entrancy	✓	✓	✓	✓	✓	✓	×	6
	Unchecked math	×	×	×	✓	×	✓	×	2
	Exposed functions	×	✓	✓	×	×	✓	×	3
	'tx.origin' usage	✓	✓	✓	✓	×	✓	×	5
	'blockhash' usage	✓	✓	×	✓	×	×	×	3
	Denial of Service (DoS)	×	×	×	✓	×	×	×	1
	'send' instead of 'transfer'	×	×	×	✓	×	×	×	1
	Style violation	×	×	×	✓	×	×	×	1
	Redundant fallback function	×	×	×	✓	×	×	×	1
EVM	Immutable bugs	×	✓	×	×	×	×	×	1
	Ether lost in transfer	×	✓	×	✓	×	×	×	2
Blockchain	Unpredictable state	×	×	×	×	×	×	×	0
	Generating randomness	×	×	×	×	×	✓	×	1
	Timestamp dependency	✓	×	✓	✓	×	✓	×	4
	Lack of transactional privacy	×	×	×	×	×	×	×	0
	Transaction-ordering dependency	✓	✓	×	×	×	✓	×	3
	Untrustworthy data feeds	×	×	×	×	×	×	×	4
	Sum	7	8	6	12	2	9	1	

Figure 2: Comparison of vulnerability detection capabilities of the tools based on vulnerability taxonomy

3.3 TheDAO Attack

TheDAO is an abbreviation for "The Decentralized Autonomous Organisation". The idea behind the TheDAO, and DAOs in general, is to codify the apparatus of an entire organisation in a smart contract, thus eliminating the need for documents and governance by individuals. The aim is to create an organization with decentralized control. TheDAO, and DAOs in general, work as follows [30]:

1. A group of people write the smart contract(s) that will govern the organisation.
2. This is followed by an initial funding period where people purchase tokens granting them voting rights.
3. After the funding period, the DAO begins to operate, and people can propose ideas (for TheDAO it was ideas for smart contracts) to invest in, and members with voting rights can then approve proposals.

TheDAO launched on 30th April 2016, with an initial funding period of 28 days. By the end of the period, TheDAO had raised over $150 million in ether from more than 11.000 members, making it the largest crowd funding project in history, at the time [30]. If token holders had not supported the decisions made by TheDAO, the option to opt-out and get their initial investment back would have been possible via an exit-door, called the split function. It was also possible for token holders to split off and start their own "child DAO", but the ether held by the child could not be accessed for 28 days [20]. On 12th June, one of TheDAO's creators announced that a recursive-call bug (re-entrancy) had been found in TheDAO's code but despite this, the funds held by the contract were not at risk [30]. On 17th June, an unknown attacker exploited the re-entrancy vulnerability in the split function and managed to siphon away 3.6 million ether (roughly one-third of TheDAO's funds). At the time, that was the equivalent of around $60 million. If used as intended, a user sends a request to exit TheDAO, and the split function follows two steps [20]:

1. In exchange for their DAO tokens the users get back their ether investment
2. Register the transaction in the ledger and update the token balance

Exploiting the vulnerability, the attacker made a recursive function in the request, thus modifying the behaviour as follows [20]:

1. In exchange for their DAO tokens the users get back their ether investment
2. Before the transaction can be registered, the recursive function made the code return to the start and transfer more ether for the same DAO tokens.

To show this, Figure 3 presents a simplified version of TheDAO contract, sharing some of the vulnerabilities of the original.

```
1  contract SimpleDAO {
2    mapping (address => uint) public credit;
3
4    function donate(address to){
5      credit[to] += msg.value;
6    }
7
8    function queryCredit(address to) returns (uint
         ){
9      return credit[to];
10   }
11
12   function withdraw(uint amount) {
13     if (credit[msg.sender] >= amount) {
14       msg.sender.call.value(amount)();
15       credit[msg.sender] -= amount;
16     }
17   }
18   /* additional functions and constructor below
         */
19 }
```

Figure 3: Simplified TheDAO contract [12]

With SimpleDAO, members can donate funds to contracts of their choice and they in turn can withdraw the funds invested in them. The first step an adversary must take is to publish the Mallory contract, shown in Figure 4.

```
1 contract Mallory {
2   SimpleDAO public dao = SimpleDAO(0x354...);
3   address owner;
4
5   function Mallory(){ owner = msg.sender; }
6   function() { dao.withdraw(dao.queryCredit(this
        )); }
7   function getJackpot(){ owner.send(this.balance
        ); }
8 }
```

Figure 4: Mallory contract exploiting SimpleDAO [12]

The adversary proceeds by using the donate function to donate ether to Mallory, invoking Mallory's fallback (line 6). The fallback function invokes SimpleDAO's withdraw function, transferring ether to Mallory. The function call in SimpleDAO's line 14 has the side effect of invoking Mallory's fallback again, interrupting the withdraw function before credit gets updated. Hence, the check at line 13 succeeds again. This loop continues until either: (i) all gas is exhausted, (ii) the call stack is full, or (iii) all funds have been withdrawn from SimpleDAO leading to a balance of zero [12].

3.4 Avoiding Re-entrancy

To recap, re-entrancy means that functions can be called repeatedly, before the first invocation of the function has finished, potentially resulting in malicious activity. An example of re-entrancy is shown in Figure 5.

```
1 contract Fund {
2     mapping(address => uint) shares; //Mapping
          of ether shares of contract
3
4     function withdraw() public { //Withdraw
          share
5         if (msg.sender.call.value(shares[msg.
              sender]))
6             shares[msg.sender] = 0;
7     }
8 }
```

Figure 5: Contract with re-entrancy vulnerability [31]

The balance of the caller is not reset to zero until after the caller's code is executed (line 7). As such, later invocations succeed and can potentially call withdrawBalance multiple times, as was the case in TheDAO attack [31]. The second issue to note here is the use of call, as this function forwards all remaining gas by default, giving more room for malicious behaviour when calling an external contract [32]. Send or transfer, while also triggering code, are considered to be safer variants, as gas forwarding is limited to a stipend of 2.300 gas (enough to log an event but not enough for an attack) [17]. So, in line 7 send(shares[msg.sender]) should be used instead of call.value(shares[msg.sender]). Another way to avoid re-entrancy attacks is to complete internal work first and then call the external contract [33]. An example for this is illustrated in Figure 6.

```
1 contract Fund {
2     mapping(address => uint) shares;
3
4     function withdraw() public {
5         var share = shares[msg.sender];
6         shares[msg.sender] = 0;
7         msg.sender.transfer(share);
8     }
9 }
```

Figure 6: Avoiding re-entrancy vulnerability [31]

This is also called the *Checks-Effects-Interaction pattern*. The name stems from how function code can be ordered to avoid certain side-effects or unwanted execution. First, preconditions are handled (in line 6), then changes to the contract state are handled (in line 7), and external contracts are interacted with (in line 8) [32].

4 Discussion

The taxonomy shown in Figure 1 (based on [12] and [16]) is arguably currently the most valid representation of security vulnerabilities in smart contracts as the categorization covers all possible levels of vulnerabilities by taking the entire ecosystem (Solidity, EVM, and blockchain) of the Ethereum blockchain into account. As common for security vulnerabilities in general, they are presented with appended severity levels, i.e. the risk vulnerability carries and the overall damage they can potentially cause. However, categorizing smart contract vulnerabilities based on severity alone is insufficient, as it gives no indication about the specific nature of vulnerabilities and their root cause.

Even though the root causes for the occurrence of several severe smart contract vulnerabilities are discussed in [2], only technical factors are covered; but this is actually insufficient in any environment with human involvement. A closer look at the incidents reveals that eventually a large number of vulnerabilities in smart contracts occur because of developer mistakes. Furthermore, taking into account the immutability of a blockchain classical mitigation of vulnerabilities is actually not enough – they have to be avoided. Hence, avoidance of vulnerabilities in smart contracts means secure development in form of a security by design development cycle. This includes among others a well thought design as well as adhering to best-practice patterns (such as the Checks-EffectsInteractions pattern for re-entrancy) and employing code analysis tools for security checks.

Looking at the code analysis tools, results of this research work showed that now security tools just cover a certain number of vulnerabilities. Some, such as Remix and Security focus on severe vulnerabilities only, while others, such as SmartCheck, Mythril, and Oyente, feature a more comprehensive set of vulnerability detection capabilities. Meanwhile Gasper and F* focus on only one or two very specific vulnerabilities, although in a thorough way. As already argued previously, low and moderate

vulnerabilities should be included in categorizations and scientific research in general.

The majority of the tools are still in beta, several shortcomings were identified (e.g. in [16]) and there is much room for improvement, such as in the user interfaces for visualization of results. Some of the missing features are highlighting the line containing vulnerable code, explaining detected vulnerabilities, proposing a reference solution and showing simple examples, providing additional resources for each vulnerability, providing an option to mark false positives, showing the total sum of detected vulnerabilities. The most important feature would be a comprehensive overview, which security vulnerabilities can be detected and if they apply to the (Solidity) source code as well as EVM byte-code or to only one of them. Such an overview would be helpful for research, but currently this is only partially done by SmartCheck [27], Remix [26], and in the paper introducing Oyente [19].

5 Conclusion

In the wake of Bitcoin's introduction, and blockchain in general, Ethereum and its implementation of a smart contract environment is a relatively new platform with its launch only three years ago. There is an extensive amount of room for further improvements when it comes to the detection and mitigation of security vulnerabilities in smart contracts, such as further effort to taxonomise them, automated test environments once the code analysis tools are out of beta, or closing the research gap of zero-day vulnerabilities in smart contracts. Moreover, last but not least the awareness of developers for the importance of a secure development process.

REFERENCES

[1] S. Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. Retrieved July 18, 2018 from https://bitcoin.org/bitcoin.pdf.

[2] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen. 2018. A survey on the security of blockchain systems, CoRR, vol. abs/1802.06993, arXiv: 1802.06993. Retrieved July 18, 2018 from http://arxiv.org/abs/1802.06993.

[3] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, 2017. Blockchain challenges and opportunities: A survey. International Journal of Web and Grid Services, December 2017.

[4] V. Buterin. 2014. A next generation smart contract & decentralized application platform, White Paper, 2014. Retrieved August 18, 2018 from https://github.com/ethereum/wiki/wiki/White-Paper

[5] CoinMarketCap. 2018. Cryptocurrency market capitalizations, Retrieved February 27, 2018 from https://coinmarketcap. com/

[6] M. Alharby and A. van Moorsel. 2017. Blockchain-based smart contracts: A systematic mapping study. CoRR, vol. abs/1710.06372, 2017. arXiv: 1710 . 06372. Retrieved August 18, 2018 from http://arxiv.org/abs/1710.06372.

[7] N. Szabo. 1997. Formalizing and securing relationships on public networks. First Monday, vol. 2, no. 9, 1997, ISSN: 13960466. DOI: 10.5210/fm.v2i9.548. Retrieved February 27, 2018 from http://journals.uic.edu/ojs /index.php/fm/ article/view/548.

[8] C. Dannen. 2017 Introducing Ethereum and Solidity, Foundations of Cryptocurrency and Blockchain Programming for Beginners. 1st ed. Apress, 2017, ISBN: 978-1-48422535-9. DOI: 10.1007/978-1-4842-2535-6.

[9] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia. 2017. Dissecting ponzi schemes on ethereum: Identification, analysis, and impact. CoRR, vol. abs/1703.03779, 2017. arXiv: 1703.03779. Retrieved February 27, 2018 from http://arxiv.org/abs/ 1703.03779.

[10] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth, and G. Rosu. 2017. Kevm: A complete semantics of the ethereum virtual machine. Retrieved February 27, 2018 from https://www.ideals.illinois.edu/bitstream/handle/2142/97207/hildenbrandt-saxena-zhu-rodrigues-guth-daian-rosu-2017-tr.pdf

[11] K. Christidis and M. Devetsikiotis. 2016. Blockchains and smart contracts for the internet of things. IEEE Access, vol. 4, pp. 2292–2303, 2016. DOI: 10.1109/ACCESS. 2016.2566339.

[12] N. Atzei, M. Bartoletti, and T. Cimoli. 2017. A survey of attacks on ethereum smart contracts sok. In Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204, New York, NY, USA: Springer-Verlag New York, Inc., 2017, pp. 164– 186, ISBN: 978-3-662-54454-9. DOI: 10.1007/978-3662-54455-68. https://doi.org/10.1007/978-3-662-54455-68.

[13] M. Bartoletti and L. Pompianu. 2017. An empirical analysis of smart contracts: Platforms, applications, and design patterns. CoRR, vol. abs/1703.06322, 2017. arXiv:1703.06322. Retrieved July 18, 2018 from http://arxiv.org/abs/ 1703.06322.

[14] T. Cook, A. Latham, and J. H. Lee. 2017. Dappguard: Active monitoring and defense for solidity smart contracts. Retrieved July 18, 2018 from https://courses.csail.mit.edu/6.857/2017/project/23.pdf

[15] T. Chen, X. Li, X. Luo, and X. Zhang. 2017. Underoptimized smart contracts devour your money," in 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Feb. 2017, pp. 442–446. DOI: 10.1109/SANER.2017.7884650.

[16] A. Dika. Ethereum smart contracts: Security vulnerabilities and security tools. 2017. Master Thesis, Norwegian University of Science and Technology, Retrieved February 27, 2018 from https://brage.bibsys.no/xmlui/bitstream/handle/11250/2479191/18400_FULLTEXT.pdf?sequence=1&isAllowed=y

[17] GitHub. (2018). Ethereum contract security techniques and tips. Retrieved July 18, 2018 from https://github.com/ethereum/wiki/wiki/Safety

[18] F. Zhang, E. Cecchetti, K. Croman, A. Juels, and E. Shi. 2016. Town crier: An authenticated data feed for smart contracts. In Proceedings of the 2016 ACM SIGSAC, ser. CCS '16, Vienna, Austria: ACM, pp. 270– 282, ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978326. http://doi.acm.org/10.1145/2976749.2978326.

[19] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. 2016. Making smart contracts smarter. In Proceedings of the 2016 ACM SIGSAC, ser. CCS '16, Vienna, Austria: ACM, 2016, pp. 254–269, ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978309. http://doi.acm.org/10.1145/2976749.2978309.

[20] A. Rosic. 2017. What is ethereum classic? ethereum vs ethereum classic, [Online]. Retrieved April 5, 2018 from https://blockgeeks.com/guides/what-is-ethereumclassic/

[21] Solidity. 2018. Security considerations. Retrieved April 2, 2018 from http://solidity.readthedocs.io/en/latest/security-considerations.html

[22] ConsenSys. 2018. Ethereum smart contract best practices, Recommendations for smart contract security in solidity, Retrieved April 2, 2018 from https://consensys.github.io/smart-contract-best-practices/recommendations/

[23] K. of the Ether Throne. 2018. Contract safety and security checklist, Retrieved April 3, 2018 from http://www.kingoftheether.com/contract-safety-checklist.html

[24] Ethereum. 2018. Ethereum, Retrieved April 3, 2018 from https://www.ethereum.org/

[25] Securify. 2018. Formal verification of ethereum smart contracts, Retrieved April 7, 2018 from https://securify.ch/

[26] Remix. 2018. Web-based user interface, Retrieved April 7, 2018 from https://remix.ethereum.org/

[27] SmartCheck. 2018. Web-based user interface, Retrieved April 7, 2018 from https://tool.smartdec.net/

[28] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella, 2016. Formal verification of smart contracts. In Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, ser. PLAS '16, Vienna, Austria: ACM, 2016, pp. 91–96, ISBN: 978-1-4503-4574-3. DOI: 10.1145/2993600.2993611. http://doi.acm.org/10.1145/2993600.2993611.

[29] B. Mueller. Github: Mythril, [Online]. Available: https://github.com/ConsenSys/mythril (visited on 07/04/2018).

[30] D. Siegel. 2016. Understanding the dao attack, Retrieved April 5, 2018 from https://www.coindesk.com/understanding-dao-hack-journalists/

[31] Solidity. 2018. Solidity documentation, Retrieved January 4, 2018 from http://solidity.readthedocs.io/en/latest/

[32] M. Woehrer and U. Zdun. 2018. Smart contracts: Security patterns in the ethereum ecosystem and solidity. In 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE), Mar. 2018, pp. 2–8. DOI: 10.1109/IWBOSE.2018.8327565.

[33] GitHub. 2018. Ethereum smart contract best practices, Known attacks, Retrieved April 2, 2018 from https://consensys.github.io/smart-contract-best-practices/knownattacks/