School of Computation, Information and Technology — Informatics

Technische Universität München

Bachelor's Thesis in Information Systems

Automated Smart Contract Quality Assurance

Metin Lamby

School of Computation, Information and Technology — Informatics

Technische Universität München

Bachelor's Thesis in Information Systems

Automated Smart Contract Quality Assurance

Automatisierte Qualitätssicherung von Smart Contracts

Author:	Metin Lamby
Submission Date:	15.06.2023
Main Supervisor:	Valentin Zieglmeier
Second Supervisor:	Christian Ziegler

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.06.2023 Metin Lamby

Abstract

Smart contract exploitability highlights its requirements for code security. Audits are performed to identify vulnerabilities, while automatic detection tools are employed to facilitate the process. This paper addresses the automatic detection of Ethereum and Algorand smart contract vulnerabilities using static analysis. To establish the solution, we analyze critical attributes of the Ethereum and Algorand platforms and identify common vulnerabilities based on literature reviews. We study and extend the static analysis frameworks Tealer and Slither to automatically detect the existence of frozen tokens, generating randomness, denial of service, and centralization risk vulnerabilities in Transaction Execution Approval Language (TEAL) and Solidity contracts. Finally, we evaluate our implementation and describe business applications and future works.

Contents

1. Introduction

The current financial system has many inefficiencies, including low asset liquidity, high barriers to entry into asset classes like art and real estate [61] and transparency. Decentralized Finance (DeFi), a blockchain-based financial infrastructure, challenges these inefficiencies, specifically through tokenization. The idea of tokenization is "to make assets more accessible and transactions more efficient" [70, p. 5]. Most tokens are issued on a blockchain through a smart contract. In the context of DeFi, smart contracts refer to small applications deployed on a blockchain that facilitate the storage and allocation of crypto-assets [70].

Tokenization entails opportunities as well as risks for investors. On the one hand, existing financial products and services can be improved by increasing the transparency of asset ownership and raising liquidity through fractionalization and subdivisions [48]. On the other hand, fund-governing smart contracts might contain vulnerabilities that may result in token holders losing their investments. For example, US$ 6.6 million were exploited from the SoarCoin[1] smart contract, which included a *centralization risk* vulnerability. Smart contract audits are hired to assess the technical quality of applications. However, these may cost up to US$ 15,000 on average [37] and are time-consuming. Efforts have emerged to automate the smart contract quality assurance process to reduce time and costs.

We perform two separate literature reviews on Ethereum and Algorand smart contract vulnerabilities. As a result, we provide an overview of vulnerabilities that exist in both platforms, also highlighting issues specific to either Ethereum or Algorand. In addition, we describe the static analysis framework designs of Slither and Tealer. To the best of our knowledge, we deliver the first analysis of the Tealer framework, as the tool has yet to be leveraged in academic publications. Our main contribution is the extension of both frameworks by implementing multiple vulnerability detectors. The addressed contract issues include *frozen tokens, generating randomness, Denial of Service (DoS), and centralization risks*. Finally, we explain use cases of automated smart contract quality assurance tools to increase the security and efficiency of workflows stakeholders employ.

[1]SoarCoin: `https://etherscan.io/address/0xd65960facb8e4a2dfcb2c2212cb2e44a02e2a57e`

2. Related Work

2.1. Overview of Vulnerabilities in Solidity Smart Contracts

Ethereum has a peak Total Value Locked (TVL) of $USD 145 billion, which accounts for 57% of the TVL in all DeFi ecosystem [100]. As a result, academic research has grown on Ethereum smart contract vulnerabilities. Zhou et al. [100] present a standardized reference framework for systematically evaluating and comparing DeFi incidents. The analysis involves the examination of 77 academic papers, 30 audit reports, and 181 real-world incidents. In this study, the authors uncover disparities between academic researchers and practitioners in the DeFi community. Chen et al. [23] provide a systematic and comprehensive survey on Ethereum systems security. In total, 40 types of vulnerabilities at different layers of the Ethereum architecture were identified, and root causes were described. Other notable papers that provide an overview of Ethereum smart contract vulnerabilities include the works of Kushwaha et al. [51], Dika and Nowostawski [26], and Tikhomirov et al. [78].

2.2. Overview of Vulnerabilities in TEAL Smart Contracts

In contrast to the considerable body of research on Ethereum smart contract vulnerabilities, Algorand lacks systematic research in terms of security [66]. Angelis et al. [13] assess the security of five blockchain platforms, including Algorand. The evaluation includes smart contract security. Due to a lack of academic publications, a study of the practitioners' community, including audit reports and GitHub repositories might be beneficial, an approach utilized by Zhou et al. [100].

2.3. Detection of Solidity Smart Contract Issues

Academic papers suggest implementations to automatically detect Solidity vulnerabilities based on static and dynamic analysis. Static analysis examines code without execution, while dynamic analysis tools rely on execution of the contract [32].

Focusing on static analysis, related works include multiple design choices along the implementation design. Next to Ethereum smart contract source code being utilized as input data [78, 68, 34, 32, 54, 23, 56], solidity byte code can also be leveraged [87, 55, 85, 81]. All related tools internally transform the input data into Intermediate Representation (IR)s. Amongst the related works studied, the most common IRs are XML Parse Tree [78], Abstract Syntax Tree (AST) [68, 34, 32, 54], Control Flow Graph (CFG) [87, 55, 81, 56], and Static Single Assessment (SSA) [32, 85].

Similarly, different static analysis techniques are used, including hybrid approaches between static analysis and machine learning [34, 54], pattern matching techniques [68, 78], symbolic execution [87, 55, 81], control flow analysis [32], and a combination of datalog, data flow, and control flow analysis [85, 56]. Finally, all studied tools provide the user with a bug report that summarizes the output of their vulnerability detection tools.

We describe the four most related works in greater detail. Brent et al. [19] provide an analysis pipeline that converts low-level Ethereum Virtual Machine (EVM) bytecode to semantic logic relations. In Vandal, the security analysis is expressed in a logic specification written in the Soufflé language, leveraging a datalog analysis technique. Luu et al. [55] propose a framework to detect vulnerabilities based on symbolic execution. Here, the values of program variables are represented as symbolic expressions of the input symbolic values. Each symbolic path has a path condition that must satisfy for execution to follow that path, assessing the soundness of a contract. Feist, Grieco, and Groce [32] convert a Solidity smart contract into an IR that uses a reduced instruction set to ease the implementation of analyses without losing semantic information that would be lost in transforming Solidity to bytecode. Finally, Tsankov et al. [85] symbolically analyze a contract's dependency graph to extract semantic information. Later, it checks for patterns that capture conditions for proving if a security property holds true.

2.4. Detection of TEAL Smart Contract Issues

To the best of our knowledge, there are no academic publications on TEAL vulnerability detection techniques. However, the publicly available tool Tealer[1] provides a framework to detect vulnerabilities. Tealer is a static analyzer for Algorand smart contracts and provides detector features for a few vulnerabilities. The tool is not based on an academic paper and does not provide clear documentation. The authors of Tealer, namely Trail of Bits,[2] are the same who developed the Slither static analyzer for Solidity smart contracts. The tool parses a contract and builds its CFG, which the analysis is based on.

[1]Tealer: `https://github.com/crytic/tealer`
[2]Trail of Bits: `https://www.trailofbits.com/`

3. Background

To provide some context, we briefly summarize the fundamental concepts of blockchain, smart contracts, and tokens relevant to our research.

3.1. Blockchains

A permissionless blockchain is a distributed and public ledger that is practically immutable being maintained by a decentralized (i.e., without a central repository such as banks, companies, or government) peer-to-peer network so that all committed transactions are stored in a chain of blocks [38, 97, 88]. Transactions are "cryptographically signed instructions from accounts" [30] that append data to the underlying blockchain. Blockchains utilize append-only properties, in which transactions added to the ledger cannot be modified [67, p. 1]. Different types of transactions are contained in the blocks comprising the respective ledger. For example, a transaction on the Ethereum network can be a transfer of funds, a smart contract deployment call, or a contract execution call.

3.2. Smart Contracts

In general, smart contracts are software applications consisting of code (functions) and data (state) that are deployed to smart contract-enabling blockchain networks (e.g. Ethereum and Algorand) in the form of a transaction [97, p. 32]. After deployment, smart contracts are identified with a unique address so users can send transactions to the contract address [55]. Transactions trigger the execution of contract logic such as calculations, information storage, or fund allocation [97, p. 32]. Looking at the current application domain of smart contracts, they are mostly experimented with when trying to simplify business logic between parties without the need for a middleman [49].

In the financial industry, public blockchain networks leveraging smart contracts can be used "to conduct transactions without having to rely on centralized service providers such as custodians, central clearinghouses, or escrow agents" [71].

3.3. Tokens

In general, tokens can represent any form of value, including a firm's loyalty program, casino chips, or claims to drinks in festivals [60]. In the context of blockchain, token systems also have various use cases, including sub-currencies representing assets like USD or gold, company stocks, secure unforgeable coupons, and point systems for incentivization [20]. Often, tokenization is associated with DeFi and plays a crucial role in supporting new financial infrastructure. Blockchain tokens are smart contracts that facilitate the storage and allocation of crypto-assets [70] without needing an intermediary. Using this technology, financial products and services can be improved by faster settlement, lower costs, increased transparency of asset ownership, and raising liquidity through fractionalization and subdivisions [48].

4. Ethereum and Algorand

Developers utilize the Ethereum and Algorand platforms to develop Decentralized Applications (Dapps). Nevertheless, they differ in many aspects, which we describe below.

4.1. Platforms

Since 15 September 2022, the Ethereum network leverages the Proof of Stake (PoS) consensus mechanism [44]. Ethereum utilizes a Turing-complete distributed computation engine, the EVM, on which smart contracts are deployed and executed [93].

Algorand is an open-source, public blockchain [45, p. 15] that uses a Pure Proof of Stake (PPoS) consensus mechanism. The platform aims to solve the blockchain trilemma: scalability, decentralization, and security. Like Ethereum, Algorand leverages a distributed computation engine, the Algorand Virtual Machine (AVM), that runs on every network node and executes smart contracts [13]. In addition, the Algorand blockchain is non-forking [45, p. 15]. Hence, transactions are considered final as soon as they are executed and included in a block. Smart contracts in Algorand can be split into two categories, namely *stateless* and *stateful*. Stateful smart contracts, or applications, are deployed to the blockchain and approve logic that manipulates its stored state [45, p. 17]. Only Algorand stateful smart contracts will be subject to our study as they are the equivalent of Ethereum smart contracts.

Both Ethereum and Algorand are account-based [7] blockchains. An account associated with a user and not a contract is an *Externally Owned Account (EOA)* in Ethereum and an *account* in Algorand [7]. Furthermore, the unique identifier with which a contract is identifiable on the blockchain is a *contract account* in Ethereum and corresponds to an Algorand *application identifier* [7]. Like Ethereum smart contracts, Algorand applications can make transactions from their application accounts. These transactions are called *inner transactions* on Algorand [7].

4.2. Smart Contract Architecture

Regardless of the platform, smart contracts are primarily responsible for implementing the logic associated with a distributed application.

The anatomy of smart contracts differs amongst both platforms. Firstly, in Algorand, smart contracts are implemented in the *ApprovalProgram* and the *ClearStateProgram*. The ApprovalProgram is responsible for processing all application calls to the contract, except the *clear* call. Most of the logic of a distributed application written for the Algorand ecosystem is implemented in the ApprovalProgram. The ClearStateProgram removes the smart contract from the user's balance record [5]. On the other hand, Ethereum smart contracts programs are inspired by classes of object-oriented languages [79, p. 6] and are not split into two separate programs.

Data in an Ethereum smart contract must either be stored in *memory* or *storage*. Data that persists outside of a function call and is therefore stored on the blockchain itself is referred to as *storage* and is represented by state variables. The datatypes of state variables must be declared explicitly and include most types that also exist in object-oriented programming, including *boolean*, *integer*, and *byte arrays*. Data that is only required during the execution of a function is stored in *memory* [28]. Contrary, Algorand applications have three different types of storage, namely *local storage*, *global storage*, and *box storage* [6]. If data is supposed to be stored in an account's balance record, *local storage* is used. Like Ethereum storage data, Algorand *global storage* is data stored on the blockchain for the contract globally. Finally, *box storage* is also stored on the blockchain, allowing contracts to use larger segments of storage [5]. Algorand smart contracts only utilize *unsigned 64-bit integers* and *byte strings* [11].

Ethereum *functions* can be *internal* and *external*. Internal functions can only be accessed from within the current contract or contracts deriving from it. External functions can be called from other contracts and by sending transactions to the respective contract by using the contracts interface. However, external functions cannot be called internally. In addition, Ethereum application functions can be *public* or *private*. Public functions can be called from within the contract or externally. Private functions are only visible for the contract defined in and not in derived contracts. Ethereum smart contracts also enable the usage of the `view` modifier, which, if declared, does not include state-modifying behavior in the function body. Finally, built-in functions often cover standard smart contract behavior, like sending Ether [28], the native cryptocurrency of the Ethereum blockchain. In contrast, Algorand smart contracts are written in assembly-like language programs, which are processed one line at a time, pushing and popping values on and off a stack [11]. However, when compiled, Algorand applications do not include the concept of a function as it exists in Ethereum smart contracts.

4.3. Programming Languages

In Algorand, smart contracts are written in TEAL. The Turing-complete assembly-like language is processed by the AVM [11], a bytecode-based stack interpreter that executes programs associated with Algorand transactions [10]. TEAL provides a set of operators that operate on the values within the stack. Many programs use the `txn` opcode to reference the current transaction's list of properties to process the provided data in the encoded logic of the application [11]. A TEAL smart contract snippet is illustrated in listing 4.1.

```
int 0
txn ApplicationID
==
bz not_creation
byte "Creator"
txn Sender
app_global_put
```

Listing 4.1: An example of an assembly-like TEAL program [3]

Ethereum smart contracts are primarily written in Solidity, an object-oriented, high-level language. In contrast to assembly-like TEAL, Solidity is a curly-bracket language influenced mainly by C++ and is statically typed [29]. Solidity smart contracts are compiled into EVM bytecode, where the EVM is a stack-based virtual machine that executes a sequence of bytecode instructions. EVM bytecode consists of low-level machine language opcodes, such as `ADD`, `SUB`, `PUSH`, and `JUMP` [16]. Each represents an operation that can access data, retrieve blockchain information, or interact with other accounts [62]. Solidity provides developers with a variety of Application Programming Interface (API)s, some specific to smart contracts, to implement logic [58]. A Solidity smart contract snippet can be seen in listing 4.2.

```
function sendFunds(address receiver, uint amount) public {
    require(amount <= balances[msg.sender], "Insufficient balance.");
    balances[msg.sender] -= amount;
    balances[receiver] += amount;
    emit Sent(msg.sender, receiver, amount);
}
```

Listing 4.2: An example of a smart contract written in high-level Solidity language [29]

Algorand smart contracts can also be written in a high-level language using a Python library for generating TEAL programs, namely PyTEAL. However, our research on smart contract vulnerabilities focuses on TEAL. This is because PyTEAL applications are compiled into TEAL programs which are executed by the AVM. We study the final program structure before the compilation to the respective bytecode. Hence, TEAL is the subject of our study on Algorand and Solidity is the subject of our study on Ethereum. As smart contract programming languages play a crucial role in developing decentralized applications, addressing the associated security concerns that may affect whole ecosystems is imperative.

4.4. Security Concerns

Since DeFi is a developing, interdependent, and open ecosystem, it is challenging to provide security to stakeholders, causing many incidents in the DeFi space. A DeFi incident refers to a series of actions taken within the ecosystem that result in an unexpected financial loss to stakeholders [100]. According to Zhou et al. [100], stakeholders suffered a total loss of at least US$ 3.24 billion after interacting with the DeFi ecosystem from Apr 30, 2018 to Apr 30, 2022. The most common DeFi incident cause is smart contract vulnerabilities [100, 23].

Literature suggests a handful of points on why the implementation of smart contracts might be particularly prone to errors in the DeFi stack. Several factors contribute to smart contract vulnerabilities, including but not limited to: bugs present in the smart contract code [69], developers' limited application domain knowledge [25, p. 80], risks associated with composability [64], potential mismatch between the semantics of smart contract programming languages and programmer intuition [15, p. 165], and the immaturity of smart contract compilers [101]. Most importantly, smart contracts are immutable by design, resulting in a lack of vulnerability-patching mechanisms during development.

Because the blockchain leverages the append-only property, changes to the ledger data are recorded in new blocks rather than modified in old blocks [38]. Because a permissionless blockchain is an immutable and publicly accessible ledger, it allows users to analyze the entire transactional history of a blockchain, unlike traditional databases where data entries can be overridden [97, p. 2]. The auditability property of the blockchain increases trust in the ledger [38] because all parties can refer to the same data, improving system traceability and transparency [88]. Even though the tamper-evident and tamper-resistant design of a blockchain provides many advantages, it is still a highly controversial concept [38].

If the data appended to the ledger is erroneous, it cannot be reverted after transaction finality [38]. Transaction finality is the assurance or guarantee that blockchain transactions "cannot be altered, reversed, or canceled after they are completed" where transactions are not automatically or instantly final, but the probability of transactional finality increases as more blocks are appended to the ledger [40]. Since a transaction on smart contract-enabling networks can be smart contract deployment calls, these programs are also immutable by design, preventing any updates to the business logic once the contract is deployed and transactional finality is assured [31]. Smart contracts, like all software applications, are error-prone. Nevertheless, smart contract development is the opposite of agile [65].

Not only are token contracts immutable by design, token contracts are generally designed to allocate and hold funds. Besides all the advantages smart contracts and the underlying immutable ledger provides, they are not a secure platform yet [69]. This makes them very tempting attack targets, as a successful attack may allow the attacker to steal funds from the contract directly [63]. Therefore, smart contract vulnerability research is crucial for enhancing security and trust in decentralized applications.

5. Smart Contract Vulnerabilities

To better understand the issues that may occur during the development and interaction with smart contracts, we perform two separate literature reviews on Ethereum and Algorand smart contract vulnerabilities.

5.1. Ethereum Smart Contract Vulnerabilities Systematic Literature Review

The systematic literature review was conducted using a predefined search strategy using publicly available databases, including IEEE Xplore and ACM Digital Library. The search terms used were built with a query of the form (“Ethereum”) AND (“smart contracts”) AND (“vulnerabilities” OR *synonyms*). The synonyms used in the query included “risks”, “issues”, and “concerns”. In addition, our selection criteria were studies published in the English language between 2015 and 2023. The studies were further screened based on their relevance to smart contract quality assurance. We filtered for papers related to our research through a title and abstract review and removed duplicates, leaving us with 106 remaining works.

5.2. Algorand Smart Contract Vulnerabilities Unsystematic Literature Review

Initially, we utilized a similar systematic literature as previously. The search terms used were built with a query of the form (“Algorand”) AND (“smart contracts”) AND (“vulnerabilities” OR *synonyms*). Again, the synonyms used in the query included “risks”, “issues”, and “concerns”. The publicly available database IEEE Xplore did not return any results. Contrary, ACM Digital Library returned 48 results. However, after further screening all 48 academic papers based on their relevance to the topic of Algorand smart contract quality assurance, we concluded that none of the papers were relevant to our problem statement. Therefore, we used an unsystematic literature review process that neglected a predefined methodology and selection criteria of works under investigation. We mainly use Google Scholar as our primary source of information, next to GitHub repositories, blog articles, and smart contract audit reports. Focusing on academic papers, we marked eight relevant papers, which were gathered after using the *Advanced Search* feature in Google Scholar with the same query as in our systematic approach. We hypothesize that Google Scholar returns more relevant resources compared to our systematic approach because Google Scholar indexes a wide range of academic sources beyond ACM and IEEE publications including articles, conference papers, theses, and preprints. Also, the Google search engine might provide resources that have not yet been formally published in ACM Digital Library or IEEE Xplore, therefore having access to more data.

5.3. Findings

We derive the following insights from our literature review.

5.3.1. Ethereum Smart Contract Vulnerability Research Outpaces Algorand

After conducting a literature review on Ethereum, we found 106 relevant works. However, our systematic review on the Algorand platform yielded no relevant papers. This suggests that there is a greater focus on researching smart contract vulnerabilities in Ethereum compared to Algorand. This might be due to Ethereum being a more mature platform than Algorand. The whitepaper of Ethereum was published in 2014 [20], while the whitepaper of Algorand was published in 2016 [24]. In addition, as Ethereum is the most widely used blockchain platform [51], more research might emerge from a wider audience than in the Algorand community.

5.3.2. There are more known Ethereum Smart Contracts Vulnerabilities

This might be a natural effect of our first finding in 5.3.1. From our literature review, we gathered 41 known vulnerabilities that might emerge during the development and interaction with Ethereum smart contracts. However, it is essential to exercise caution when interpreting the quantity of known vulnerabilities. Different papers often use different names to describe the same vulnerability (e.g. *weak sources of randomness* and *randomness using "Block Hash"*). In addition, many vulnerabilities have relationships with each other. For example, the vulnerability of missing permission verification is a parent issue of the vulnerabilities *unprotected self-destruct* and *unprotected ether withdrawal*. Due to the lack of public research on Algorand application vulnerabilities, we find a smaller number of issues. To the best of our knowledge, Angelis et al. [13] provide the only public academic resource that provides information on Algorand smart contract vulnerabilities. In total, they identify five vulnerabilities that might emerge during the development of Algorand smart contracts. An overview of all known vulnerabilities, including their description, we identified during our literature review on both platforms can be found in the appendix A.

5.3.3. Smart Contract Vulnerabilities: Ethereum vs. Algorand

The Algorand smart contract vulnerabilities suggested by Angelis et al. [13] might also exist in Ethereum smart contracts, namely *frozen tokens*, *insufficient signature information*, *generating randomness*, *token lost to orphan address* and *unprotected suicide*. However, some vulnerabilities are specific to a platform. The *reentrancy* vulnerability is specific to Ethereum applications because Algorand smart contract calls are "allowed one way only, thus if smart contract A calls a smart contract B, the latter cannot call back A" [13]. Hence, an application cannot make an application call to itself [7]. In addition, the *integer overflow and underflow* vulnerability does not occur in Algorand contracts as TEAL natively solves under or overflow issues [13]. Further Ethereum contract specific issues include: *transaction ordering dependence*, *under-priced opcodes*, *short address* and *erroneous visibility* [13]. On the other hand, the Algorand *rekeying* vulnerability suggested by the non-academic source provided by Trail of Bits [83] does not occur in Ethereum applications because there is no direct equivalent of the functionality to allow "account holder to maintain a static public address while dynamically rotating the authoritative private spending key(s)" on Ethereum [7]. Finally, the *missing group size check* vulnerability [83] also does not occur in Ethereum applications.

5.3.4. Focusing on Specific Vulnerabilities: Concentration of Ethereum Vulnerability Detection Tools

We find that in Ethereum, the vulnerability detection tools suggested in academia often focus on the same vulnerability, as seen in figure 5.1. While our systematic literature review suggests that 66 tools aim to detect the *reentrancy* vulnerability, there are very few or no detection tools for vulnerabilities, including *backdoor threats*, *missing zero address validation* and *local variable shadowing*. The addressed concentration might reflect the fatality of a potential exploit of the respective vulnerability, suggesting that *reentrancy* is more critical to detect than a *replay attack*. Nevertheless, this finding should be the subject of future work to confirm the suggested explanation.

5.3.5. Awareness Gap in Academia on Centralization Risks

Since only five Algorand smart contract vulnerabilities were identified from an academic source during our literature review, we expand our research on GitHub repositories,

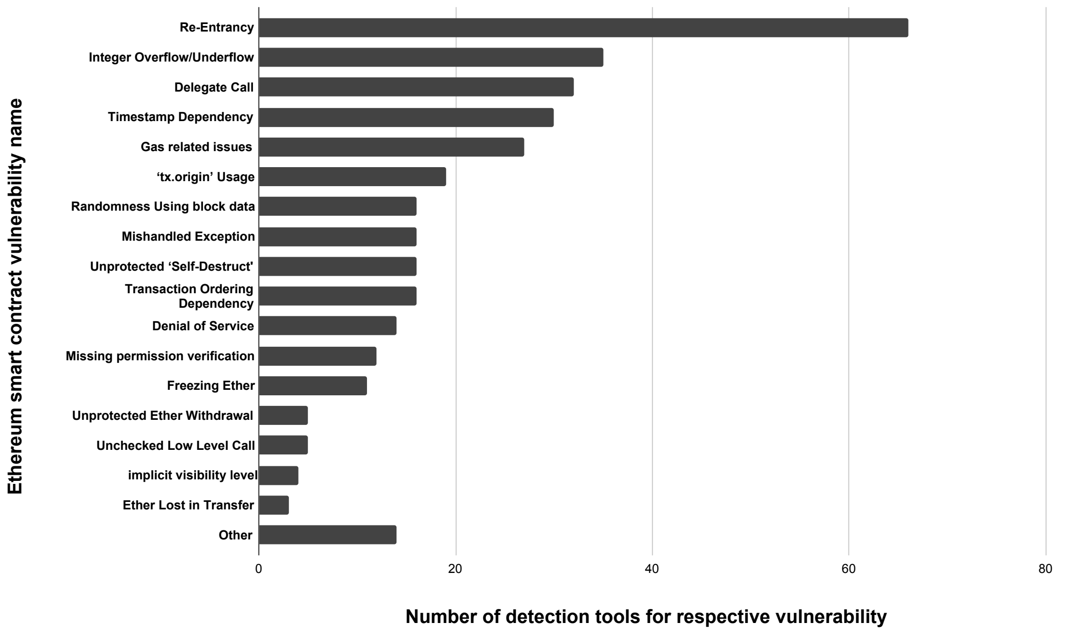

Figure 5.1.: Ethereum smart contract vulnerabilities and the number of detection tools for the respective vulnerability as suggested in academia

blog articles, and smart contract audit reports. The smart contract auditor Certik[1] published five audit reports on projects written in an Algorand smart contract-specific language, namely PyTEAL, and TEAL. We find that in all projects, the auditor identifies *centralization-related risks* with major severity. Interestingly, this vulnerability was not mentioned in any academic paper found on Algorand smart contract vulnerabilities and was only addressed by one paper in our systematic literature review on Ethereum, namely [56]. However, Ma et al. [56] coin the issue as *backdoor threats*. The lack of academic research on this vulnerability suggests low relevance. However, significant players from the practitioners' community, such as Coinbase[2] and Certik, highlight the importance of *centralization risks*. The Centralized Exchange (CEX) reviews all to-be-listed tokens for this issue type before allowing the underlying token to be traded [77]. Centralization issues were the most common attack vector exploited in 2021 [22]. According to the report, US$ 1.3 billion in user funds were lost across 44 DeFi hacks due to user privileges in 2021. We find that there is an awareness gap in academia on smart contract *centralization risks*. Before aiming to improve smart contract security with static analysis we define vulnerabilities in scope.

5.4. Vulnerabilities in Focus

We select vulnerabilities for detection based on three criteria. Firstly, the vulnerability should exist in Solidity and TEAL applications. Because of the lack of academic research on Algorand smart contract vulnerabilities, we also take into account vulnerabilities suggested by non-academic sources. After filtering vulnerabilities based on these criteria, we consider all items in Table 5.1 as candidates. We group related vulnerabilities as they often describe the same problem. For example, we group the *unprotected suicide* issues reported by Angelis et al. [13] with *access control* issues as reported by [83]. We drop those vulnerabilities that have been addressed by detectors written for both platforms, avoiding redundancy. We filter out vulnerabilities if they are not detectable with static analysis. An adequate validity check for addresses used in smart contracts can only be prevented by manually assuring the correctness of the address [23, p. 13]. Therefore, no automated solutions, including static analysis techniques are sufficient to detect the *token lost to orphan address* vulnerability. Hence, we do not consider this issue.

After our selection process, we decide to program static analysis detectors for *frozen tokens, generating randomness, DoS, and centalization risks* vulnerabilities for both TEAL and Slither smart contracts.

[1]Certik: `https://www.certik.com/`
[2]Coinbase: `https://www.coinbase.com/`

Table 5.1.: Smart contracts vulnerabilities that might occur in Ethereum and Algorand contracts

Name	**Description**	**Reference Ethereum**	**Reference Algorand**
Frozen token	The ability of users to deposit funds to a contract with the inability to withdraw the funds from those accounts, effectively freezing their funds [23, p. 9].	[23, 95, 47, 91, 35]	[13]
Generating randomness	This vulnerability addresses smart contracts using Pseudorandom Number Generators (PRNG) to create random numbers [13].	[23, 51, 26, 95, 53]	[13]
Token lost to orphan address	There is a lack of validation checks on payment transactions regarding the recipient's address [23, p. 12].	[23]	[13]
Access controls	The vulnerability is caused by inadequate authentication enforced by a contract on critical contract functionality (e.g. destroy contract) [23, p. 10].	[23, 51, 53, 27, 41]	[83, 13]
Denial of service	Occurs if the execution of a transaction is reverted due to a thrown error or a malicious callee contract that deliberately interrupts the execution [13].	[23, 26, 2, 78, 43]	[83, 13]
Centalization risks	High-privilege functions, which can only be invoked by certain group of accounts [56].	[56]	[21]

6. Static Analysis Frameworks

In our implementation, we leverage the static analysis technique to detect patterns in TEAL and Solidity that contribute to a potential vulnerability. We take advantage of open-source static analysis frameworks and extend those with our detection scripts. As discussed in 2, there are multiple open-source static analysis frameworks for Solidity applications, including [19, 55, 32, 85]. On the other hand, there is only one open-source framework that we know of for TEAL smart contracts, namely Tealer. Due to the lack of alternatives, we leverage the Tealer static analysis framework for TEAL contract vulnerability detectors. Since the Slither [32] framework was developed by the same authors as Tealer, namely Trail of Bits, we use it for Solidity contract vulnerability detectors. In addition, Slither allows us to write vulnerability detection scripts by identifying pre-defined source code patterns gathered during our literature review without processing them to a higher abstraction level, a technique leveraged by Schneidewind et al. [73] and Brent et al. [19].

6.1. Slither

Slither analyzes Solidity applications using static analysis in a procedure of multiple stages. A user can service the tool with a flat smart contract (i.e. the input contract must be contained in a single .sol file) that is internally compiled into the EVM bytecode and further transformed into an AST represented JSON [32]. An AST is a tree aimed at representing the abstract syntactic structure of the source code. All tree nodes correspond to constructs or symbols of the underlying source code. However, unlike source code, ASTs are abstract and remove program details such as punctuation and delimiters. Still, ASTs are used to describe the lexical information and the syntactic structure of source code [99], such as the Solidity access control modifier `onlyOwner`. All further Slither processes are based on the AST format of the underlying Solidity smart contract. First, Slither recovers information about the source code, such as the contract's inheritance graph, the CFG, and the list of expressions [32]. Next, the internal representation language of Slither, SlithIR, is generated from the gathered data. SlithIR uses SSA "to facilitate the computation of a variety of code analyses" [32, p. 9]. In the last stage, Slither computes a set of analyses that provide information to the other modules, providing an API[1] to users. An overview of the framework is illustrated in figure 6.1.

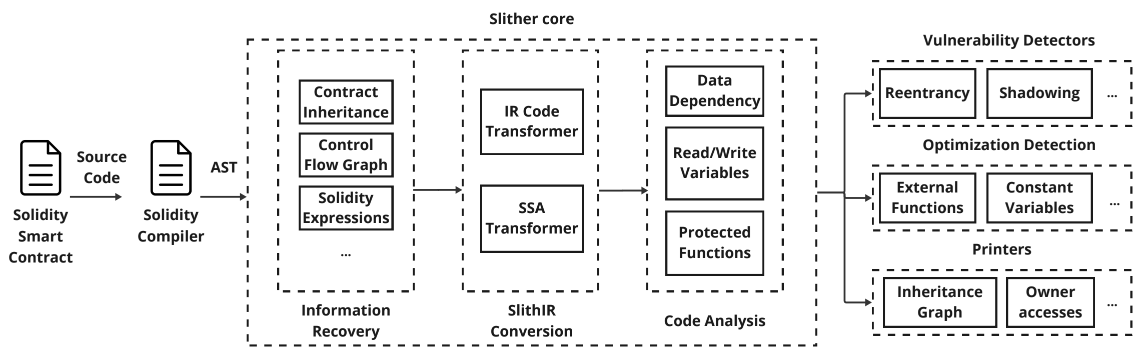

Figure 6.1.: Slither static analysis framework [32]

The Python API describes methods and objects available for custom analyses. Initially, a `slither` object has to be initialized from a Solidity source code file. A `slither` object includes an attribute `contracts`, which contains a list of contract objects derived from the source code file. A `contract` has a `name`, `functions`, and `modifiers`, among other objects. Further data can be derived from a `function`, `modifier`, and `node` object, where a CFG node contains the individual expressions of the source code program. Figure 6.2 visualizes a Slither CFG and its contained data, including Solidity source code expressions and their translation to the internal representation language SlithIR.

6.2. Tealer

Contrary to Slither, the TEAL smart contract static analysis framework is not based on an academic publication. In addition, the publicly accessible Tealer GitHub repository does not provide detailed documentation on its architecture and API. We analyze the framework's source code of the GitHub repository to understand the tool.

Tealer takes the TEAL source code of a smart contract as input and parses it. Each line of the source code is parsed as a TEAL instruction used to construct basic blocks and other rules, such as subroutines. After parsing, the basic blocks containing TEAL instructions comprise a CFG of the contract. Based on the parsed code, vulnerability detectors, and printers can be utilized by the end user. Vulnerability detectors are Python scripts that examine the CFG and search for syntactical patterns in the program. An overview of the Tealer implementation design is illustrated in 6.3.

[1]Slither API: `https://crytic.github.io/slither/slither.html`

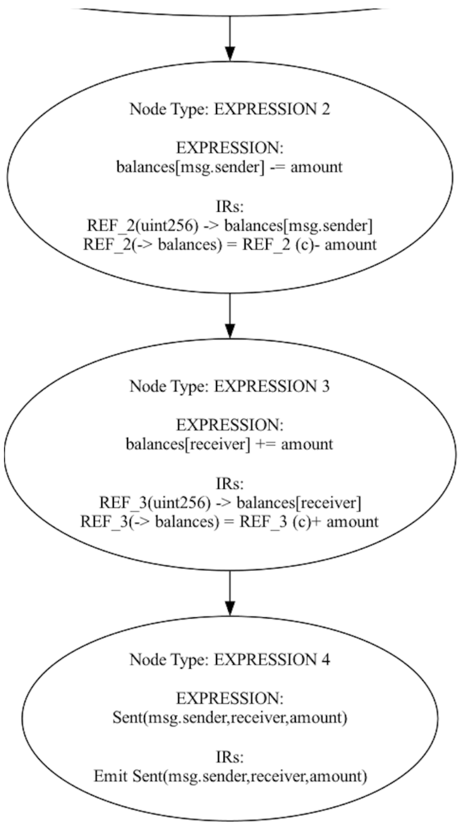
Node Type: EXPRESSION 2
EXPRESSION:
balances[msg.sender] -= amount
IRs:
REF_2(uint256) -> balances[msg.sender]
REF_2(-> balances) = REF_2 (c)- amount
Node Type: EXPRESSION 3
EXPRESSION:
balances[receiver] += amount
IRs:
REF_3(uint256) -> balances[receiver]
REF_3(-> balances) = REF_3 (c)+ amount
Node Type: EXPRESSION 4
EXPRESSION:
Sent(msg.sender,receiver,amount)
IRs:
Emit Sent(msg.sender,receiver,amount)

Figure 6.2.: Our example Solidity smart contract's "Send" function [29] represented as Slither nodes including its expressions and SlithIR translation in a CFG format.

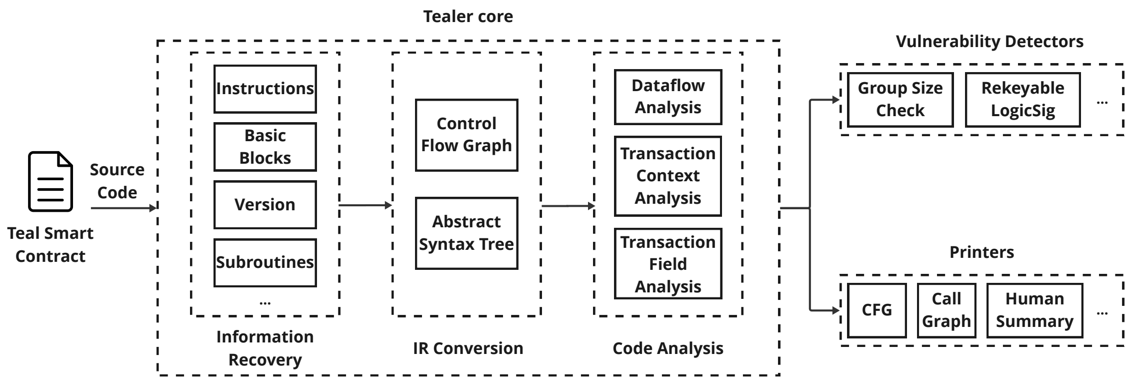

Figure 6.3.: Tealer static analysis framework

Tealer also provides a list of APIs to facilitate the extension of vulnerability detectors. A TEAL contract is represented as a `teal` object. A `teal` object contains a list of `subroutines` and basic blocks that contain source code instructions. Figure 6.4 visualizes part of a Tealer CFG and its contained data, including basic blocks and source code instructions.

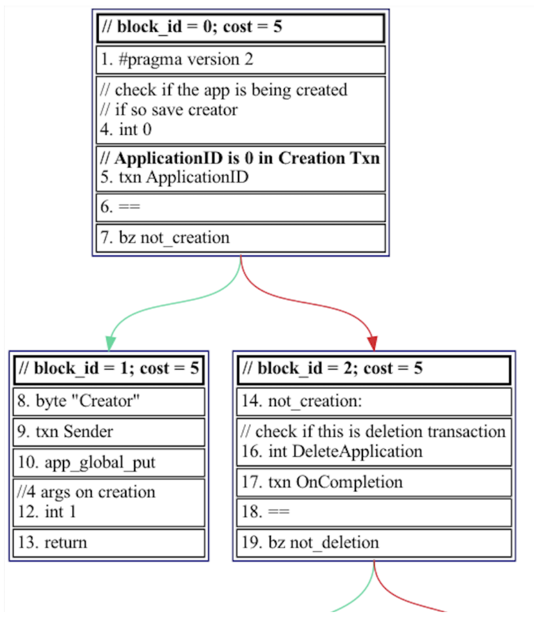

Figure 6.4.: Snippet of example TEAL smart contract [3] represented as Tealer CFG consisting of basic blocks made up of source code instructions.

We leverage the Slither and Tealer APIs to identify the existence of *frozen tokens, generating randomness, DoS, and centalization risks* in Solidity and TEAL smart contracts respectively.

6.3. Comparing Implementation Designs

After Slither takes Solidity source code files (`.sol`) as input, it uses its dependencies to the Solidity compiler to parse the AST [32], which is later used for information recovery. On the other hand, Tealer takes TEAL source code files (`.teal`) as input. In theory, Algorand applications written in PyTEAL could also be analyzed. However, a compiler would have to transform the PyTEAL source code into TEAL before it is analyzed, as Tealer only analyzes TEAL code. In addition, the TEAL code is not converted into an AST before being utilized by the Tealer core module. Slither might leverage an AST at an early stage while Tealer does not because Solidity is a high-level language [29] whose syntax includes punctuation and delimiters, as illustrated in 4.2, which are removed when converted into an AST [99]. Contrary, TEAL is an assembly-like language [11], as illustrated in 4.1, which might not require further abstraction. Also, Slither utilizes the recovered information to convert the provided source code into its hybrid IR SlithIR, which facilitates the implementation of analyses without losing critical source code semantic information [32]. Later code analysis implemented as part of the Slither core module is based on SlithIR. On the contrary, Tealer does not leverage its own IR or SSA. Tealer code analyses are based on IRs of a CFG and an AST which is constructed by Tealer itself, without a dependency on a third-party compiler.

7. Automated Smart Contract Vulnerability Detection

Both static analysis frameworks allow us to detect source code patterns in TEAL and Solidity [32] that might make up a vulnerability. We extend both frameworks by adding Python detection scripts that include pre-defined patterns to the vulnerability detection module illustrated in 6.3 and 6.1. To identify the critical patterns, we use data collected during the literature review in 5.1 and 5.2. If required because of a lack of academic references, especially for TEAL patterns, we also use non-academic sources as we did previously.

Because the depth of a static analysis depends on the patterns included in the examination [52], we highlight that critical patterns potentially making up a vulnerability are not limited to the ones we include in our study. We only cover the patterns identified in our literature review. Therefore, false negatives might occur during the detection [52], which we will discuss further in our evaluation. A simplified version of the detectors is illustrated in algorithms 1 and 2. Next to the difference in detected patterns, the critical difference between Tealer and Slither detection scripts is the accessibility of source code expressions. In Slither, Solidity expressions are analyzable via the nodes of a Slither CFG, as described in 6.2. In Tealer, TEAL instrcutions are decomposable via the basic blocks of a Tealer CFG as decribed in 6.4.

Algorithm 1 Detecting Solidity functions that include vulnerability pattern

```
$vulnerableFunctions \leftarrow []$
for $contractFunctions$ do
    for $functionCFGNodes$ do
        if Expression in Slither CFG node contains vulnerability pattern then
            $vulnerableFunctions \leftarrow function$
        end if
    end for
end for
```

Algorithm 2 Detecting TEAL expressions that include vulnerability pattern

```
$vulnerableBasicBlocks \leftarrow []$
for $contractCFGBasicBlocks$ do
  for $bbInstructions$ do
    if Instruction in Tealer CFG basic block contains vulnerability pattern then
      $vulnerableBasicBlocks \leftarrow bbInstruction$
    end if
  end for
end for
```

7.1. Frozen Tokens

The *frozen tokens* vulnerability has many synonyms, including *asset frozen* [91] and *greedy contracts* [35]. Based on our literature review dataset on Ethereum smart contract vulnerabilities, we find 11 detection solutions suggested in academia. The vulnerability occurs in smart contracts when they can receive funds but cannot release them [35]. In other words, a deposit functionality exists while a withdrawal functionality does not [23, p. 9]. The lack of an outbound fund traffic logic is primarily due to two reasons. Firstly, contracts may not implement such logic at all [91, 35]. Secondly, the logic is being delegated to an external contract that does not deliver its intended service, for example, because it is destructed [23, p. 9] [35]. The issue can be prevented by assuring that deposit-enabling contracts do not outsource withdrawal functions to another contract [23, p. 9]. A greedy contract vulnerability caused 150 thousand worth of ether to be stolen due to the *asset frozen* issue and caused the loss of approximately 300 thousand dollars [91].

7.1.1. Solidity Patterns and Slither Detector

The Solidity patterns described in 7.1 only make up a vulnerability if the deposit pattern exists in a smart contract while no withdrawal pattern is implemented. Deposit functions are marked with a `payable` keyword, meaning users can send ether to the contract by calling this function [85]. Withdrawal logic is implemented using either `send()`, `call()`, or `transfer()`, which are built-in functions provided by Solidity.

Since Slither already includes a detector for the *frozen tokens* vulnerability, we do not extend the tool to avoid redundancy. The *locked-ether* Slither detector[1] studies submitted Solidity smart contracts by analyzing the source code patterns described in 7.1.

[1] Slither Locked Ether Detector Implementation: `https://github.com/crytic/slither/blob/master/slither/detectors/attributes/locked_ether.py`

Table 7.1.: Solidity deposit and withdrawal patterns useful for frozen token vulnerability detection

Pattern	**Reference**	**Implementation**

deposit	[92, 47, 87, 85]	`function functionName() ... payable ...{`
		`    ...`
		`}`

withdrawal with transfer	[92, 47, 78, 76]	`function functionName() ... {`
		`    address.transfer(...);`
		`}`

withdrawal with send	[92, 47, 78]	`function functionName() ... {`
		`    address.send(...);`
		`}`

withdrawal with call	[47, 78]	`function functionName() ... {`
		`    address.call{...};`
		`}`

7.1.2. TEAL Patterns and Tealer Detector

The process of transferring funds on Algorand is similar whether they are Algos, the currency used in Algorand [45, p. 15], or Algorand Standard Assets (ASA) [7], a layer-1 asset of the Algorand blockchain [45, p. 15]. A fund reallocation is handled by transactions submitted to the blockchain addressing the underlying smart contract. While a *Payment* transaction handles a transfer of Algos to an account, a relocation of ASAs is managed by an *AssetTransfer* transaction. In practice, the type of transaction can be specified using the TxType field [12]. In addition, Algorand transactions can be grouped before submitting them simultaneously, while the transaction types included in the group can vary. For example, a *Payment* transaction can be grouped with an *AssetTransfer* transaction to exchange Algos for an ASA [45, p. 16].

We point out that the transactions critical for detecting a *frozen token* vulnerability in Algorand are not encoded within the smart contract, only the transaction evaluation is [57]. This impacts our definition of a *frozen token* vulnerability in Algorand. As discussed in 7.1.1, deposit and withdrawal logic is implemented into a Solidity smart contract. On the other hand, deposit and withdrawal logic in Algorand is managed by *AssetTransfer* or *Payment* transactions, which are generated outside of a smart contract and, therefore, not included in the application source code and not detectable using static analysis. Since only transaction validity checks can be encoded into an Algorand smart contract, we argue for a *frozen token* definition differentiation between both platforms. The Solidity *frozen token* vulnerability should be referred to as *missing deposit validity verification* and *missing withdrawal validity verification* in TEAL.

Since Matteo [57] provides the only academic source in our literature review dataset that includes a source code pattern description of a deposit and withdrawal evaluation, we leverage its logic. However, all patterns were implemented in PyTEAL. Therefore, we compile a dummy contract and inject the described pattern to generate a TEAL equivalent. The simplified version of PyTEAL payment evaluation, as illustrated by Matteo [57], and its TEAL translation generated through a compilation process are described in 7.1 and 7.2, respectively.

```
alice = Addr("...")
bob = Addr("...")
valid_payment = And(
 Gtxn[1].type_enum() == TxnType.Payment,
 Gtxn[1].receiver() == alice,
 Gtxn[1].amount() == Int(0),
 Gtxn[1].sender() == bob,
)
```

Listing 7.1: PyTEAL snippet of a simplified payment transaction evaluation [57]

```
gtxn 1 TypeEnum
int pay
==
gtxn 1 Receiver
addr ...
==
&&
gtxn 1 Amount
int 0
==
&&
gtxn 1 Sender
addr ...
==
&&
```

Listing 7.2: Compiled TEAL code of Listing 7.1

In Algorand, a deposit pattern can be differentiated from a withdrawal pattern by analyzing a transaction's *sender* and *receiver* attributes. A deposit's sender is the sender of the (grouped) transaction, while a deposit's receiver can be an arbitrary address. On the other hand, a withdrawal's sender can be an arbitrary address, while a withdrawal's receiver is the sender of the (grouped) transaction [57].

Because we want to detect missing validation on payment transaction fields, we do not consider a differentiation between deposit and withdrawal transactions in our detection script. We consider a smart contract as vulnerable that includes the evaluation of a payment transaction within a group of transactions where the transaction attributes, including *amount, receiver, sender, and type*, were not validated.

7.2. Centralization Risks

As mentioned in section 5.3, we find that there is an awareness gap in academia on *centralization risks*. Still, because significant players from the practitioners' community highlight the issue's importance, we include this vulnerability in our analysis. Due to the lack of academic research on this issue, we define the *centralization risk* based on the definition of Ma et al. [56] and the definitions used in the practitioners' community. Firstly, Ma et al. [56] coins *centralization risks* as *backdoor threats* and define them

Table 7.2.: TEAL deposit and withdrawal evaluation patterns useful for missing validity verification detection

Pattern	Reference	Implementation

deposit validation within a transaction group	[57, 45]	`gtxn 1 TypeEnum`
		`int pay`
		`==`
		`gtxn 1 Sender`
		`gtxn 0 Sender`
		`==`
		`&&`
		`gtxn 1 Receiver`
		`byte "Escrow"`
		`app_global_get`
		`==`
		`...`

withdrawal validation within a transaction group	[57]	`gtxn 1 TypeEnum`
		`int pay`
		`==`
		`gtxn 1 Receiver`
		`gtxn 0 Sender`
		`==`
		`&&`
		`gtxn 1 Sender`
		`byte "Escrow"`
		`app_global_get`
		`==`
		`...`

as “threats related to high-privileged functions” [56, p. 2]. In addition, the smart contract auditing firm Certik classifies the vulnerability with *major* severity. It defines findings related to centralization as application logic that acts against the nature of decentralization, including “specialized access roles in combination with a mechanism to relocate funds” [21]. Finally, the CEX Coinbase defines the addressed issue as *superuser risks* and flags their occurrence when “single actors have the sole authority to execute a dangerous function” [77]. As a result, we detect *centralization risks* in TEAL and Solidity smart contracts when a contract includes fund modifying logic where the access to that logic is restricted to privileged users with access control patterns.

Note that the addressed *access control* vulnerability in 5.4 caused by inadequate authentication enforced by a contract on critical contract functionality [23, 51, 53, 27, 41] clashes with the *centralization risk* definition. We argue that even though access control appears to be a logical initiative in an open ecosystem, it may also cause *centralization risks* in a decentralized ecosystem. Therefore, a conflict of interest might occur depending on the stakeholders involved. Privileged access to critical functions creates a dependency of the underlying system on a small set of accounts which may lead to undesirable consequences for stakeholders, as described in the complementary GitHub repository of SmartCheck [74]. This dependency becomes a risk if the private key of the privileged user‘s address becomes compromised, allowing a malicious third party to take control over critical contract functionality. An example of an undesirable consequence for stakeholders resulting from attacked access controls is the artificial dilution or inflation of the value of a token implemented as a smart contract [18]. As a result, we argue that access controls in smart contracts may threaten the validity of the general definition of a smart contract used by some parties. According to Atzei, Bartoletti, and Cimoli [15, p. 164], smart contracts contain functionality enforced by the blockchain‘s consensus mechanism *“without relying on a trusted authority”*. However, if access control patterns are implemented on critical functions, *smart contract users certainly do rely on a trusted authority*, the privileged users. Hence, centralized access control can introduce new trust assumptions and degrade security [17], where the risk does occur from an implementation error but from social risks, namely malicious behavior, that affects the underlying contract security. In summary, we consider privileged user roles as a potential vulnerability. However, if avoided by reducing access controls, this vulnerability may introduce other information flow issues, such as making contracts *self-destructible* [18].

An example of an exploited *centralization risk* is the Ethereum smart contract of SoarCoin. Here, the `zero_fee_transaction()` function shown in listing 7.3 on line five contains the `onlycentralAccount` modifier, defined on lines one to four, that restricts the access to the fund modifying function to the privileged user who at the moment of execution has the `central_account` role. The illustrated function gives the user with the `central_account` role the privilege to take any token from any account [56], creating a *centralization risk*. As defined by the ERC20 token standard, a transfer process from one address to another should get permission from the sender's address. However, no approval verification was implemented in the underlying function to permit the transfer operation [56]. An attacker could steal $US 6.6 million by exploiting the `zero_fee_transaction()` function of the Soar Labs smart contract because of the described issue [56].

```
modifier onlycentralAccount {
    require(msg.sender == central_account);
    _;
}
function zero_fee_transaction(...) onlycentralAccount returns(...) {
    if (...) {
        balances[_from] -= _amount;
        balances[_to] += _amount;
        Transfer(_from, _to, _amount);
        return true;
    } else {
        return false;
    }
}
```

Listing 7.3: Example of a centralization risk found in the Ethereum SoarCoin smart contract

7.2.1. Solidity Patterns and Slither Detector

There are multiple Solidity source code patterns with which *access control* to contract functions can be implemented, as illustrated in table 7.3. Regardless of the pattern used, Solidity access control implementations usually check whether the global variable `msg.sender`, which allows contract developers to read the sender's address of the current contract call, satisfies certain conditions [96].

In Solidity, the most common access control patterns leverage the built-in `require` function within a modifier or at the beginning of a function. Modifiers are enclosed code units that enrich functions to modify their flow of code execution [92]. A typical use case for modifiers is to check certain conditions before executing the function, for example, for authorization purposes [92]. The modifier concept helps replace recurring checks with a single keyword, increasing readability [72]. The described balance modification pattern in 7.3 uses the `add(...)` function to increase the user's

Table 7.3.: Solidity centralization risk related vulnerability patterns

Pattern	**Reference**	**Implementation**

onlyOwner modifier	[82, 72, 18, 56]	`modifier only_owner {`
		`    require(msg.sender == address(...));`
		`    ...`
		`}`
		`function functionName(...) only_owner ... {`
		`    ...`
		`}`

require within function	[96, 80, 72]	`function functionName() ... {`
		`    require(address(...) == msg.sender)`
		`    ...`
		`}`

if statement within function	[72]	`function functionName() ... {`
		`    if (msg.sender == address(...)) {`
		`        ...`
		`    }`
		`}`

balance modification	[92, 32]	`mapping(address => uint) balances;`
		`function functionName(...) ... {`
		`  balances[...] = balances[...].add(...);`
		`}`

balance. This function is provided by the `SafeMath` library.[2] However, this function can also be replaced by an arithmetic operation, which we also include in our detection script.

7.2.2. TEAL Patterns and Tealer Detector

We take advantage of non-academic sources to evade the lack of TEAL patterns described in academia. Firstly, the official Algorand GitHub repository[3] provides TEAL contracts which we study to identify access control and balance modification patterns in TEAL. In addition, Matteo [57] implements an authentication logic in PyTEAL which we compile to a TEAL equivalent.

Table 7.4.: TEAL centralization risk patterns

Pattern	Reference	Implementation

address comparison with assert	[1, 57]	`byte "manager"`
		`app_global_get`
		`txn Sender`
		`==`
		`assert`

address comparison with branch opcodes	[3, 4]	`byte "Creator"`
		`app_global_get`
		`txn Sender`
		`==`
		`...`
		`bz failed`

balance modification	[3]	`...`
		`byte "MyBalance"`
		`...`
		`app_local_put`

[2]SafeMath: `https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/math/SafeMath.sol`

[3]Algorand GitHub: `https://github.com/algorand/smart-contracts`

The typical Solidity pattern that enables the smart contract accounting of balances `mapping (address => uint) public balances;` is often replaced by local storage in Algorand [7]. TEAL contracts usually implement access control using two control flow patterns. Firstly, the `assert` opcode immediately fails unless the stack expressions located above the `assert` opcode return a non-zero number [9]. Secondly, the conditional branching opcodes `bz target` and `bnz target` can be utilized. While the `bz target` opcode branches to `target` if the previously evaluated condition on the stack returns zero, the `bnz target` opcode branches to `target` if the previously evaluated condition on the stack returns non-zero [9]. The `bz target` opcode was leveraged in 6.4. If `txn ApplicationID` were to be zero, indicating that the Algorand application was deployed before the evaluated transaction, the application logic branches to `target not_creation:`. On the other hand, if `txn ApplicationID` were to equal one, indicating that the application does not exist on the ledger yet, the application logic follows the original dataflow and enters the Tealer CFG basic block with `block_id = 1`.

7.3. Generating Randomness

According to our systematic literature review, the Solidity *generating randomness* vulnerability is an issue for which many detection solutions have been implemented in academia. Based on our data set, 16 implementations address this issue, excluding Slither. In addition, Tealer does not provide a detector to identify this issue in TEAL applications. The vulnerability might occur in a contract if the notion of randomness is implemented as part of the application logic. Often, private seeds are used as PRNG to generate a pseudorandom number [23, p. 13]. However, these sources of randomness are somewhat predictable [53], which malicious actors can manipulate to attack the vulnerable function. For example, in gambling contracts, hackers may use this to increase their winning rate [95]. As a best practice, external sources of randomness should be used [51]. The Ethereum SmartBillions contract uses a predictable source of randomness, leading to attackers capturing 400 Ether from the application [86].

7.3.1. Solidity Patterns and Slither Detector

In Solidity, there are multiple patterns in how block data is used to generate pseudorandom numbers, as illustrated in 7.5.

Even though Slither already includes a similar detector, namely its weak-prng detector[4], it only includes pattern identifiers for `block.timestamp`, `now`, or `block.blockhash`. Because our literature review dataset suggests further patterns to generate randomness,

[4]Slither weak-prng detector: https://github.com/crytic/slither/wiki/Detector-Documentation

Table 7.5.: Vulnerable sources of randomness in Solidity

Pattern	**Reference**	**Implementation**

block hash	[51, 50, 26, 86]	`function functionName() ... {`
		`    return blockhash(...);`
		`}`

block number	[50, 95, 86, 41]	`function functionName() ... {`
		`    return block.number;`
		`}`

block timestamp	[95, 86, 41]	`function functionName() ... {`
		`    return block.timestamp;`
		`}`

block coinbase	[86]	`function functionName() ... {`
		`    return block.coinbase;`
		`}`

block difficulty	[86]	`function functionName() ... {`
		`    return block.difficulty;`
		`}`

we extend the existing detector with additional patterns. In Solidity, the `block .difficulty` expression only exists in EVM versions previous to the Paris update. It behaves as a deprecated alias for `block.prevrandao` for later EVM versions. The `block.prevrandao` expression returns a random number provided by the beacon chain for EVM versions greater or equal to the Paris update (EIP-4399[5]) [75].

7.3.2. TEAL Patterns and Tealer Detector

The *generating randomness* vulnerability may also occur in TEAL contracts [13]. However, to the best of our knowledge, no academic paper lists source code patterns that describe the issue. Therefore, we utilize official documentation provided by Algorand. In Algorand, the *block seed* is a pseudorandom number generated by each block where the value is created from multiple pieces of information from the underlying ledger. The *block seed* itself should not be used as a source of randomness because it is possible to influence the value of the *block seed* by running a participation node [46]. The *block seed* can be utilized in a TEAL [9] contract using the pattern described in 7.6. The `block BlkSeed` expression only exists in AVM versions greater than seven which can be indicated with the `#pragma version 7` expression at the beginning of a TEAL program.

Table 7.6.: Vulnerable sources of randomness in TEAL

Pattern	**Reference**	**Implementation**
block seed	[46]	`block BlkSeed`

We do not find any argument in our literature review data set that argues against static analysis to detect this vulnerability because block data patterns can be identified clearly. However, we find that static analysis might not be the correct technique to detect this implementation bad practice. This is because of the challenge of implementing context awareness in static analysis, a limitation addressed by Tsankov et al. [85] who leverage an optimization technique to improve analysis precision by introducing context sensitivity. To illustrate our argument, we use the example of a *randomness using 'Block Hash'* vulnerability provided by Kushwaha et al. [51] in which the block hash pattern value is returned by a function named `randomNumber()`. Still, whether or not the example function is used within a randomness context cannot be analyzed statically. If we were to detect a *generating randomness* vulnerability based on the occurrence of

[5]EIP-4399: `https://eips.ethereum.org/EIPS/eip-4399`

block data usage only, a high false positive rate would be the result. Therefore, many papers rephrase the vulnerability and neglect the randomness context.

While Dika and Nowostawski [26] highlight that *blockhash* usage is not to be used on crucial components, Jiang, Liu, and Chan [42] and Wang et al. [87] consider an underlying contract as vulnerable with a *block information dependency* if block data is used within the same execution path as an Ether transfer in Solidity contracts. Therefore, we increase the depth of our detection script in Tealer and Slither to detect the *block information dependency* [42, 87] vulnerability, even though it is not explicitly mentioned in our literature review data set on Algorand smart contract vulnerabilities. We do so in our Slither detector by checking whether block data was utilized within the same function as one that implements the logic to transfer funds using the patterns listed in 7.1. On the other hand, as mentioned in 7.1.2, transfer logic is not encoded within applications but managed by Algorand transactions. Therefore, we modify our Tealer detector to check whether TEAL block data opcodes were used within the same execution path as a TEAL balance modifying behavior pattern described in 7.4. Suppose we identify block information usage and a write operation to local storage. In that case, we perform a graph traversal of the Tealer CFG to identify common execution paths where the starting nodes are all basic blocks, including a `block BlkSeed` opcode, and the end nodes are all basic blocks, including an `app_local_put` opcode. The graph traversal is done using the *breadth-first search* technique.

7.4. Denial Of Service

Our literature review shows that the *DoS* vulnerability is a known issue in Ethereum as 14 tools address this issue. However, the *DoS* vulnerability can be defined broadly. A system disruption might be due to an external contract call error [78], block gas limits in Ethereum [51], or a failed call [51]. Nevertheless, the common definition between the many Ethereum-related explanations and the one we find for Algorand vulnerabilities [83] is an interruption due to failed calls or *DoS with unexpected revert*. In general, the issue occurs if the execution of a transaction is reverted because of a thrown error or a malicious callee contract that interrupts the execution on purpose [13, 23].

In Algorand, *DoS* due to failed calls occur because receiving an ASA requires a user to opt-in explicitly [45, p. 36]. A transaction that attempts to transfer ASA to an account that did not opt-in to that asset will fail. The scale of the issue increases if one user is not opted-in to an asset and a contract attempts to transfer assets to multiple users. In this case, the transfer operation fails for all users, causing a *DoS* error for multiple parties [83]. To mitigate such issues, verifying whether an account has opted into an asset before transferring it is crucial [83].

On the contrary, Ethereum contracts *DoS* with unexpected revert vulnerabilities might occur in two situations. Firstly, when a conditional statement includes an external call, the callee may permanently fail, preventing the caller from completing the execution [26]. Secondly, if there is an external call function within a Solidity loop, a *DoS* attack might occur [68], a contract implementation design also described as vulnerable in Slither's GitHub repository [84]. An example of a *DoS* vulnerability containing application is the GovernMental smart contract which had collected almost 1100 Ether at one point and was then susceptible to the described issue [68].

7.4.1. Solidity Patterns and Slither Detector

In Solidity, the *DoS* with unexpected revert vulnerabilities can be identified in two implementation patterns as listed in 7.7.

Table 7.7.: Solidity DoS with unexpected revert patterns

Pattern	**Reference**	**Implementation**

conditional includes external call	[26]	`if (payable.send(...)) {`
		`    ...`
		`}`

external call in loop body	[68, 32]	`for(...) {`
		`    require(payable.send(...));`
		`}`

Slither already includes a built-in detector for the *external call in loop body pattern*[6]. However, the Slither detector does not check for the *conditional includes external call* patterns. Therefore, we extend the tool with a detector addressing this pattern and leverage the SlithIR representation to check whether a function expression is a conditional that includes external calls, which are messages that create a transaction.

7.4.2. TEAL Patterns and Tealer Detector

We identify the vulnerability in TEAL contracts if a receiver opt-in verification is missing before an Algorand inner transaction sends funds to the addressed receiver. The first

[6]Slither calls-loop detector: https://github.com/crytic/slither/wiki/Detector-Documentation

relevant pattern is the *missing opt-in verification pattern*. In TEAL, the `app_opted_in` opcode can be leveraged to verify if an account is opted into an application [9]. Here, the application represents the token contract of the ASA that is transferred in a transaction. Therefore, the application that a user must be opted in before reception is the application from which the ASA is sent. The respective address can be retrieved with the TEAL `global CurrentApplicationID` [9]. Secondly, fund withdrawal functionality from within the contract can be implemented in TEAL with inner transactions [8]. An inner transaction differs from a regular transaction we described in 7.1.2. The difference lies in the party triggering the transactions. While users outside a contract trigger regular transactions, inner transactions are triggered by application accounts [7] as part of the application logic. Since an Algorand *DoS* with unexpected revert vulnerability occurs if a contract attempts to transfer assets, leveraging the push over the pull pattern, we only consider single and grouped inner transactions in our Tealer detection script. The critical TEAL patterns included in the detection script are listed in 7.8.

Table 7.8.: TEAL DoS with unexpected revert patterns

Pattern	**Reference**	**Implementation**

opt-in verification	[9]	`...`
		`byte "Receiver"`
		`app_global_get`
		`global CurrentApplicationID`
		`app_opted_in`
		`...`

inner transaction	[8, 1]	`itxn_begin`
		`int axfer`
		`itxn_field TypeEnum`
		`int ...`
		`itxn_field Amount`
		`txn Sender`
		`itxn_field Receiver`
		`itxn_submit`

8. Evaluation

This section evaluates our smart contract vulnerability detection scripts through experiments. We aim to answer the following Research Question (RQ)s:

- *RQ1. Can we detect smart contract vulnerabilities using Slither and Tealer?*
- *RQ2. How effective are the Tealer and Slither detection scripts in identifying the vulnerabilities in focus?*

8.1. Evaluation Techniques

To answer the addressed RQ, we use our literature review data to find evaluation approaches in past research. Firstly, there are publicly available labeled data sets for Solidity smart contracts. Schneidewind et al. [73] provide a data set of Ethereum smart contract addresses mapped to *safe* and *unsafe* labels. Since the labels do not represent the included vulnerability, we do not utilize this data set. In addition, Samreen and Alalfi [68] provide a data set that includes vulnerability labels. However, the *DoS* vulnerability is the only common vulnerability between our focus area and the available labels. As discussed in 7.4, the *DoS* vulnerability can be defined broadly. Hence, we focused on the *DoS with unexpected revert*, which might only represent a subset of *DoS* vulnerabilities in the data set provided by Samreen and Alalfi [68]. We observe similar limitations to our evaluation with public data sets provided by [14, 90, 33, 53]. In addition, to the best of our knowledge, there is no publicly available TEAL smart contract vulnerabilities data set.

The second data collection and labeling approach is used to compare the effectiveness of an implementation to a similar solution from past research. Here, blockchain explorers like Etherescan [1] are used to collect Ethereum smart contracts from the ledger itself. Later, akin tools are used as a baseline to compare results to the ground truth, which is limited to the effectiveness of the utilized detection tool. This approach is leveraged by [14, 78, 85, 87, 89]. For Algorand smart contracts, data can be collected using block explorers like Algoexplorer.[2] However, as discussed in 2, the only relevant

[1]Etherescan: `https://etherscan.io/`
[2]Algoexplorer: `https://algoexplorer.io/`

tool is Tealer. The framework does not cover our vulnerabilities in focus. Therefore, it cannot be used to generate a baseline for result comparison purposes.

Next, case studies can be performed. Here, a small number of vulnerable smart contracts are evaluated that gained attention from the blockchain community due to the damage caused by exploits. While Tikhomirov et al. [78] evaluate the Ethereum smart contracts coined Genesis, Hive, and Populous, Feist, Grieco, and Groce [32] utilize the DAO and SpankChain as case study subjects. The limitations of this experiment are the size of the data set as well as the lack of case studies available for Algorand.

Finally, there are evaluation approaches based on manual processes. First, manual labeling exercises are carried out to evaluate the true positives of detection tools [36, 85, 59]. Secondly, manual vulnerability injection into smart contracts [90] can be leveraged to evaluate the RQs defined above.

8.2. Experiment: Dummy Application

We utilize manual labeling and pattern injection techniques to evaluate RQ1 based on the above-described evaluation approaches and their limitations. We implement a dummy solidity contract that allows us to evaluate if Slither can be used to detect smart contract vulnerabilities using the patterns defined in 7. To avoid redundancy while communicating the idea of vulnerability pattern injection, we illustrate one injected pattern for each smart contract-enabling platform. Source code injected into a contract is indicated with + a prefix of a source code line. Removed code is denoted with a - prefix.

8.2.1. Solidity Pattern Injection

To recall, we identify a *DoS with unexpected revert* vulnerability in Solidity if we detect a *conditional includes external call* pattern. The illustrated `denialOfService()` function in 8.1 first included a conditional in which two `uint` variables were checked for similarity. After the injection, the conditional is based on an external call `payable(owner).send(msg.value)` and therefore adheres to our *DoS with unexpected revert* definition.

```
function denialOfService() public payable returns(bool returnBool) {
    - uint constantNumber = 3;
    - uint constantNumberTwo = 3;
    - if (constantNumber == constantNumberTwo) {
    + if (payable(owner).send(msg.value)) {
        return true;
    } else {
        return false;
    }
}
```

Listing 8.1: Solidity DoS with unexpected revert vulnerability injection

After running our Slither detection scripts on the experimental Solidity dummy application, we find that all four Ethereum vulnerabilities can be detected using the Slither framework with an extension of our detection scripts, as illustrated in 8.1.

Table 8.1.: Solidity ability to detect vulnerabilities in focus

Frozen Tokens	Centralization Risks	Block Information Dependency	Denial of Service
✓	✓	✓	✓

8.2.2. TEAL Pattern Injection

To demonstrate a *centralization risk* in TEAL contracts, we inject the *address comparison with assert* pattern [1, 57] into the `new_giver:` subroutine that includes a balance modification logic after the authorization pattern. Both patterns required for the vulnerability demonstration in 8.2 are illustrated individually in 7.4.

```
new_giver:
+ byte "Creator"
+ app_global_get
+ txn Sender
+ ==
+ assert
int 0 //sender
byte "MyAmountGiven"
gtxn 1 Amount
app_local_put
b finished
```

Listing 8.2: Solidity centralization related risk injection

After running our Tealer detection scripts on the experimental TEAL dummy application, we find that all four TEAL vulnerabilities can be detected using the Tealer framework with an extension of our detection scripts, as illustrated in 8.2.

Table 8.2.: Tealer ability to detect vulnerabilities in focus

Frozen Tokens	Centralization Risks	Block Information Dependency	Denial of Service
✓	✓	✓	✓

Based on our experiment, we evaluate RQ1. It is possible to detect TEAL and Solidity smart contract vulnerabilities using static analysis under the following assumptions. Firstly, the vulnerability must be correctly defined and is not a context-related issue. Secondly, all source code patterns derived from a vulnerability definition are identified and have sufficient depth regarding considered implementation options. Finally, the static analysis scripts identifying the vulnerabilities using the extracted patterns are implemented accurately. Regardless of the platform, the necessary infrastructure to detect TEAL and Ethereum vulnerabilities using static analysis is provided by Trail of Bits in Tealer and Slither.

8.3. Threats to Validity

No labeled TEAL contracts are available for benchmarking our Tealer detection results, while the data sets of labeled Solidity contracts are incomplete for the evaluation of our Slither detection scripts. Therefore, we find that the lack of evaluation data for smart contract vulnerability research is a bottleneck to the evaluation of our study and a major limitation to the progress of secure smart contract development and interaction. We argue that RQ2 cannot be evaluated at this point, especially due to the lack of available evaluation data on TEAL smart contracts and their vulnerabilities.

In addition, while evaluating RQ1, we may introduce an element of bias by manually injecting source code patterns. We note that the vulnerabilities described in 7 are not limited to the patterns we described. Since the static analysis technique is based on established source code patterns that we audit against, the effectiveness of our implementation is limited to the source code patterns considered during the analysis [52].

We have made our experimental setup available in a GitHub repository to make our results reproducible and allow checking of our code and analysis methods.[3]

[3]Instructions for reproducing detection scripts: `https://github.com/MetinLamby/AutomatedSmartContractQAScripts`

9. Business Application and Future Work

As addressed in this project, vulnerabilities in fund-governing smart contracts may result in token holders incurring an irreversible loss on their investment. As a result, stakeholders, including token exchange platforms, should perform risk analyses on the utilize smart contracts.

Currently, tokens can be traded on a CEX and Decentralized Exchange (DEX). While companies that enable the trade of tokens centrally include Coinbase, and 360X, [1] decentralized parties include Uniswap, [2] and Sushiswap.[3] A critical difference between both types of exchanges is that CEXs are regulated, and DEXs are not, affecting trustworthiness and fund security. For example, Uniswap is a leading DEX built on Ethereum, designed to facilitate automated exchange transactions between Ether and ERC-20 tokens [94]. During the listing process of tokens, Uniswap does not maintain any requirements. Since neither contract owners nor the contract itself is audited, the majority of tokens (50.14%) listed on the platform are scam tokens [94].

Traditional CEXs depend on a central authority to facilitate trading, whereas their trustworthiness is crucial in ensuring reliable trading processes. Here, a key aspect of the legitimacy of CEXs lies in the token listing process. It aims to prevent investors from allocating their funds into vulnerable contracts, which may result in token holders incurring a loss on their investment. To avoid regulatory scrutiny and reputational damage, CEXs perform risk analyses on to-be-listed token contracts. Smart contract audits are hired as the critical component of technical quality controls. Since audits may cost up to US$ 15,000 on average [37] and may take a long time to complete, only a few stakeholders can afford such initiatives. This may create barriers to entry and harm the financial health of a CEX. Therefore, efforts have emerged to automate the smart contract quality assurance process.

Static and dynamic analysis can fit the needs by automatically detecting smart contract vulnerabilities to help perform risk analyses, reducing the reliance on audits. Our work covered static analysis on TEAL and Solidity smart contracts to mitigate the vulnerabilities of *frozen tokens, block information dependency, DoS, and centralization risks*. The research can help improve the token listing process of exchanges. In theory,

[1]360X: `https://www.360x.com/`

[2]Uniswap: `https://uniswap.org/`

[3]Sushiswap: `https://www.sushi.com/`

automatic vulnerability detectors can analyze submitted tokens to flag contracts before an audit is hired, reducing the duration and costs of token listings.

In our research, we could not evaluate the effectiveness of our vulnerability detectors. The essential research gap that should be closed in the future is the lack of vulnerability labeled smart contract data sets. Having access to alternative TEAL contract vulnerability detection tools enables the generation of a baseline to compare implementation results to ground truth, a technique used to evaluate Ethereum smart contract vulnerability detection tools. This enables an effectiveness evaluation of TEAL vulnerability detection tools. In addition, the extent to which static analysis is less functional in high-level languages compared to low-level languages when detecting smart contract vulnerabilities could be studied. The amount of implementation options for a specific vulnerability pattern seems to decrease with the abstraction level of a programming language. We speculate that a sufficient static analysis on TEAL opcodes does not require the depth of a high-level language study. Moreover, our research suggests a limitation of static analyses on context-relevant issues, such as randomness. To address this limitation, future work should include alternative quality assurance techniques, including fuzzing and machine learning solutions. Finally, to make security assessments more accessible, future work could include building an interface for the vulnerability detectors we implemented. This would allow stakeholders without a technical background to perform a smart contract risk analysis before interaction.

10. Conclusion

We perform literature reviews on Ethereum and Algorand smart contract vulnerabilities. Our data suggest that there are more known vulnerabilities that may occur in Solidity smart contracts compared to TEAL. In addition, around 50 vulnerability detection tools are available for Solidity, while only one exists for TEAL contracts. Both findings might be due to Ethereum being the more mature and widely used blockchain platform. Still, we do not argue that Algorand is the more secure smart contract development platform. Furthermore, we observe a concentration of Solidity vulnerability detection tools towards a small number of vulnerabilities while the relevance of others seems to be underestimated by academia, especially *centralization risks*. After collecting and filtering all known vulnerabilities in both platforms, we concentrate our study on four vulnerabilities: *frozen tokens, block information dependency, DoS, and centralization risks*. We analyze and extend the static analysis frameworks Slither and Tealer to automatically detect those issues.

During our static analysis, we found that the differences in architecture and programming languages between the Ethereum and Algorand platform results in significantly different vulnerable source code patterns, as described in 7.1 and 7.4. Moreover, we discuss how access control mechanisms introduce user privileges into smart contracts creating sources of centralization in decentralized networks. These risks may be exploitable when private key leakages of privileged users occur, a risk that cannot be mitigated with static analysis alone (7.2). The quality assurance technique also has its limitations on vulnerabilities where context awareness is required (7.3).

Our implementation is evaluated by injecting vulnerable source code patterns into contracts. The lack of labeled smart contract vulnerability data sets, especially on Algorand applications, is the bottleneck for evaluating the effectiveness of our detection scripts. Nevertheless, applying static analysis techniques in smart contract development offers promising potential for enhancing security, reducing vulnerabilities, and promoting the adoption of secure and reliable Dapps.

Abbreviations

DeFi Decentralized Finance

TVL Total Value Locked

EVM Ethereum Virtual Machine

AVM Algorand Virtual Machine

CFG Control Flow Graph

AST Abstract Syntax Tree

PoS Proof of Stake

PPoS Pure Proof of Stake

Dapps Decentralized Applications

EOA Externally Owned Account

TEAL Transaction Execution Approval Language

PRNG Pseudorandom Number Generators

SSA	Static Single Assessment
DEX	Decentralized Exchange
CEX	Centralized Exchange
ASA	Algorand Standard Assets
IR	Intermediate Representation
DoS	Denial of Service
RQ	Research Question
API	Application Programming Interface

List of Figures

List of Tables

Listings

Bibliography

[1] AfricaCodeAcademy. *Fidelis TEAL approval program*. URL: https://github.com/africacodeacademy/fidelis-contracts/blob/main/loans/approval.teal (visited on 04/28/2023).

[2] Nurul Aida Noor Aidee et al. "Vulnerability Assessment on Ethereum Based Smart Contract Applications." In: *2021 IEEE International Conference on Automatic Control & Intelligent Systems*. IEEE, June 2021, pp. 13–18. DOI: 10.1109/I2CACIS52118.2021.9495892.

[3] Algorand Documentation. *Algorand crowd_fund.teal Smart Contract*. URL: https://github.com/algorand/smart-contracts/blob/master/devrel/crowdfunding/crowd_fund.teal (visited on 04/28/2023).

[4] Algorand Documentation. *Algorand Dex.teal Smart Contract*. URL: https://github.com/algorand/smart-contracts/blob/master/devrel/dexapp/dex.teal (visited on 04/25/2023).

[5] Algorand Documentation. *Algorand Smart Contracts*. URL: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/ (visited on 04/23/2023).

[6] Algorand Documentation. *Contract Storage*. URL: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/state/ (visited on 04/23/2023).

[7] Algorand Documentation. *From Ethereum to Algorand*. URL: https://developer.algorand.org/docs/get-details/ethereum_to_algorand/ (visited on 04/21/2023).

[8] Algorand Documentation. *Inner transactions*. URL: https://developer.algorand.org/docs/get-details/dapps/smart-contracts/apps/innertx/ (visited on 05/05/2023).

[9] Algorand Documentation. *Opcodes*. URL: https://developer.algorand.org/docs/get-details/dapps/avm/teal/opcodes/ (visited on 04/27/2023).

[10] Algorand Documentation. *The Algorand Virtual Machine (AVM) and TEAL*. URL: https://developer.algorand.org/docs/get-details/dapps/avm/teal/specification/ (visited on 04/24/2023).

[11] Algorand Documentation. *The Smart Contract Language*. URL: `https://developer.algorand.org/docs/get-details/dapps/avm/teal/?query=public&object_source=false` (visited on 04/23/2023).

[12] Algorand Documentation. *Transaction reference*. URL: `https://developer.algorand.org/docs/get-details/transactions/transactions/` (visited on 04/28/2023).

[13] Stefano De Angelis et al. "Evaluating Blockchain Systems: A Comprehensive Study of Security and Dependability Attributes." PhD thesis. 2022. URL: `https://ceur-ws.org/Vol-3166/paper02.pdf` (visited on 03/22/2023).

[14] Nami Ashizawa et al. "Eth2Vec: Learning Contract-Wide Code Representations for Vulnerability Detection on Ethereum Smart Contracts." In: *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*. New York, NY, USA: ACM, May 2021, pp. 47–59. DOI: `10.1145/3457337.3457841`.

[15] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A Survey of Attacks on Ethereum Smart Contracts (SoK)." In: *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*. Vol. 10204. Berlin, Heidelberg: Springer-Verlag, 2017, pp. 164–186. DOI: `10.1007/978-3-662-54455-6{\_}8`.

[16] Gbadebo Ayoade et al. "Smart Contract Defense through Bytecode Rewriting." In: *2019 IEEE International Conference on Blockchain (Blockchain)*. IEEE, July 2019, pp. 384–389. DOI: `10.1109/Blockchain.2019.00059`.

[17] Rob Behnke. *Designing Secure Access Control for Smart Contracts*. Nov. 2022. URL: `https://halborn.com/designing-secure-access-control-for-smart-contracts/` (visited on 02/01/2023).

[18] Lexi Brent et al. "Ethainter: A Smart Contract Security Analyzer for Composite Vulnerabilities." In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, June 2020, pp. 454–469. DOI: `10.1145/3385412.3385990`.

[19] Lexi Brent et al. "Vandal: A Scalable Security Analysis Framework for Smart Contracts." In: (Sept. 2018). URL: `https://arxiv.org/abs/1809.03981`.

[20] Vitalik Buterin. "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform." In: (2014). URL: `https://ethereum.org/669c9e2e2027310b6b3cdce6e1c52962/Ethereum_Whitepaper_-_Buterin_2014.pdf`.

[21] Certik. *Fidelis Audit Report*. URL: `https://skynet.certik.com/projects/fidelis` (visited on 04/25/2023).

[22] Certik. *The State of Defi Security 2021*. Tech. rep. URL: `https://f.hubspotusercontent40.net/hubfs/4972390/Marketing/defi%20security%20report%202021-v6.pdf` (visited on 12/06/2022).

[23] Huashan Chen et al. "A Survey on Ethereum Systems Security." In: *ACM Computing Surveys* 53.3 (May 2021), pp. 1–43. DOI: `10.1145/3391195`.

[24] Jing Chen and Silvio Micali. "Algorand." In: (July 2016). arXiv: `1607.01341`.

[25] Kevin Delmolino et al. "Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab." In: 2016, pp. 79–94. DOI: `10.1007/978-3-662-53357-4{\_}6`.

[26] Ardit Dika and Mariusz Nowostawski. "Security Vulnerabilities in Ethereum Smart Contracts." In: *2018 IEEE International Conference on Internet of Things and IEEE Green Computing and Communications and IEEE Cyber, Physical and Social Computing and IEEE Smart Data*. IEEE, July 2018, pp. 955–962. DOI: `10.1109/Cybermatics_2018.2018.00182`.

[27] Thomas Durieux et al. "Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts." In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. New York, NY, USA: ACM, June 2020, pp. 530–541. DOI: `10.1145/3377811.3380364`.

[28] Ethereum Documentation. *Anatomy of Smart Contracts*. URL: `https://ethereum.org/en/developers/docs/smart-contracts/anatomy/` (visited on 04/23/2023).

[29] Ethereum Documentation. *Smart Contract Languages*. URL: `https://ethereum.org/en/developers/docs/smart-contracts/languages/` (visited on 04/24/2023).

[30] Ethereum Documentation. *Transactions*. Nov. 2022. URL: `https://ethereum.org/en/developers/docs/transactions/` (visited on 01/09/2023).

[31] Ethereum Documentation. *Upgrading Smart Contracts*. Oct. 2022. URL: `https://ethereum.org/en/developers/docs/smart-contracts/upgrading/` (visited on 01/11/2023).

[32] Josselin Feist, Gustavo Grieco, and Alex Groce. "Slither: A Static Analysis Framework for Smart Contracts." In: *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. IEEE, May 2019, pp. 8–15. DOI: `10.1109/WETSEB.2019.00008`.

[33] João F. Ferreira et al. "SmartBugs." In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, Dec. 2020, pp. 1349–1352. DOI: `10.1145/3324884.3415298`.

[34] Zhipeng Gao et al. "SmartEmbed: A Tool for Clone and Bug Detection in Smart Contracts through Structural Code Embedding." In: *2019 IEEE International Conference on Software Maintenance and Evolution*. IEEE, Sept. 2019, pp. 394–397. DOI: 10.1109/ICSME.2019.00067.

[35] Subhasish Goswami et al. "TokenCheck: Towards Deep Learning Based Security Vulnerability Detection In ERC-20 Tokens." In: *2021 IEEE Region 10 Symposium*. IEEE, Aug. 2021, pp. 1–8. DOI: 10.1109/TENSYMP52854.2021.9550913.

[36] Neville Grech et al. "MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts." In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (Oct. 2018), pp. 1–27. DOI: 10.1145/3276486.

[37] Hedera Hashgraph. *What Is a Smart Contract Audit?* URL: https://hedera.com/learning/smart-contracts/smart-contract-audit (visited on 05/12/2023).

[38] Frank Hofmann et al. "The Immutability Concept of Blockchains and Benefits of Early Standardization." In: *2017 ITU Kaleidoscope: Challenges for a Data-Driven Society*. IEEE, Nov. 2017, pp. 1–8. DOI: 10.23919/ITU-WT.2017.8247004.

[39] Yongfeng Huang et al. "Smart Contract Security: A Software Lifecycle Perspective." In: *IEEE Access* 7 (2019), pp. 150184–150202. DOI: 10.1109/ACCESS.2019.2946988.

[40] Obasi Ifegwu. *Finality*. URL: https://academy.binance.com/en/glossary/finality (visited on 01/25/2023).

[41] Suhwan Ji, Dohyung Kim, and Hyeonseung Im. "Evaluating Countermeasures for Verifying the Integrity of Ethereum Smart Contract Applications." In: *IEEE Access* 9 (2021), pp. 90029–90042. DOI: 10.1109/ACCESS.2021.3091317.

[42] Bo Jiang, Ye Liu, and W. K. Chan. "ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, Sept. 2018, pp. 259–269. DOI: 10.1145/3238147.3238177.

[43] Mudabbir Kaleem, Anastasia Mavridou, and Aron Laszka. "Vyper: A Security Comparison with Solidity Based on Common Vulnerabilities." In: *2020 2nd Conference on Blockchain Research & Applications for Innovative Networks and Services*. IEEE, Sept. 2020, pp. 107–111. DOI: 10.1109/BRAINS49436.2020.9223278.

[44] Elie Kapengut and Bruce Mizrach. "An Event Study of the Ethereum Transition to Proof-of-Stake." In: *Commodities* 2.2 (Mar. 2023), pp. 96–110. DOI: 10.3390/commodities2020006.

[45] Gidon Katten. "Issuing Green Bonds on the Algorand Blockchain." In: (Aug. 2021). arXiv: 2108.10344.

[46] Anne Kenyon. *Randomness on Algorand*. Sept. 2022. URL: https://developer.algorand.org/articles/randomness-on-algorand/ (visited on 04/27/2023).

[47] Zulfiqar Ali Khan and Akbar Siami Namin. "Ethereum Smart Contracts: Vulnerabilities and their Classifications." In: *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, Dec. 2020, pp. 1–10. DOI: 10.1109/BigData50022.2020.9439088.

[48] Soyeon Kim. "Fractional Ownership, Democratization and Bubble Formation - The Impact of Blockchain Enabled Asset Tokenization." In: *AMCIS 2020 Proceedings* (2020). URL: https://aisel.aisnet.org/amcis2020/adv_info_systems_research/adv_info_systems_research/19/ (visited on 01/04/2023).

[49] Manjunatha G. Kukkuru. *Smart Contracts: Introducing A Transparent Way To Do Business*. URL: https://www.infosys.com/insights/digital-future/smart-contracts.html (visited on 01/10/2023).

[50] Satpal Singh Kushwaha et al. "Ethereum Smart Contract Analysis Tools: A Systematic Review." In: *IEEE Access* 10 (2022), pp. 57037–57062. DOI: 10.1109/ACCESS.2022.3169902.

[51] Satpal Singh Kushwaha et al. "Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract." In: *IEEE Access* 10 (2022), pp. 6605–6621. DOI: 10.1109/ACCESS.2021.3140091.

[52] Peng Li and Baojiang Cui. "A Comparative Study on Software Vulnerability Static Analysis Techniques and Tools." In: *2010 IEEE International Conference on Information Theory and Information Security*. IEEE, Dec. 2010, pp. 521–524. DOI: 10.1109/ICITIS.2010.5689543.

[53] Jian-Wei Liao et al. "SoliAudit: Smart Contract Vulnerability Assessment Based on Machine Learning and Fuzz Testing." In: *2019 Sixth International Conference on Internet of Things: Systems, Management and Security*. IEEE, Oct. 2019, pp. 458–465. DOI: 10.1109/IOTSMS48152.2019.8939256.

[54] Han Liu et al. "S-gram: Towards Semantic-Aware Security Auditing for Ethereum Smart Contracts." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, Sept. 2018, pp. 814–819. DOI: 10.1145/3238147.3240728.

[55] Loi Luu et al. "Making Smart Contracts Smarter." In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, Oct. 2016, pp. 254–269. DOI: 10.1145/2976749.2978309.

[56] Fuchen Ma et al. "Pied-Piper: Revealing the Backdoor Threats in Ethereum ERC Token Contracts." In: *ACM Transactions on Software Engineering and Methodology* (Aug. 2022). DOI: 10.1145/3560264.

[57] Baratella Matteo. "Decentralized Carpooling with Algorand Blockchain." PhD thesis. 2022. URL: http://157.138.7.91/bitstream/handle/10579/21548/856097-1255954.pdf?sequence=2 (visited on 03/21/2023).

[58] Tai D. Nguyen, Long H. Pham, and Jun Sun. "SGUARD: Towards Fixing Vulnerable Smart Contracts Automatically." In: *2021 IEEE Symposium on Security and Privacy*. IEEE, May 2021, pp. 1215–1229. DOI: 10.1109/SP40001.2021.00057.

[59] Tai D. Nguyen et al. "sFuzz." In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. New York, NY, USA: ACM, June 2020, pp. 778–788. DOI: 10.1145/3377811.3380334.

[60] Luis Oliveira et al. "To Token or not to Token: Tools for Understanding Blockchain Tokens." Nov. 2018. URL: https://www.zora.uzh.ch/id/eprint/157908/ (visited on 05/17/2023).

[61] Organisation for Economic Co-operation and Development (OECD). *The Tokenisation of Assets and Potential Implications for Financial Markets*. Tech. rep. 2020. URL: https://www.oecd.org/finance/The-Tokenisation-of-Assets-and-Potential-Implications-for-Financial-Markets.pdf (visited on 05/17/2023).

[62] Zhenyu Pan et al. "ReDefender: A Tool for Detecting Reentrancy Vulnerabilities in Smart Contracts Effectively." In: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security*. IEEE, Dec. 2021, pp. 915–925. DOI: 10.1109/QRS54544.2021.00101.

[63] Daniel Perez and Benjamin Livshits. "Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited." In: (Feb. 2019). arXiv: 1902.06710.

[64] Quantstamp Announcements. *DeFi's Double-Edged Sword*. Mar. 2020. URL: https://quantstamp.com/blog/defis-double-edged-sword (visited on 12/12/2022).

[65] Haseeb Qureshi. *A hacker stole $31M of Ether — how it happened, and what it means for Ethereum*. July 2017. URL: https://www.freecodecamp.org/news/a-hacker-stole-31m-of-ether-how-it-happened-and-what-it-means-for-ethereum-9e5dc29e33ce (visited on 12/06/2022).

[66] Konrad Rotkiewicz. *Best Smart Contract Platforms*. Oct. 2020. URL: https://www.ulam.io/blog/smart-contract-platforms (visited on 03/21/2023).

[67] Muhammad Saad et al. "Exploring the Attack Surface of Blockchain: A Systematic Overview." In: (Apr. 2019). arXiv: 1904.03487.

[68] Noama Fatima Samreen and Manar H. Alalfi. "SmartScan: An approach to detect Denial of Service Vulnerability in Ethereum Smart Contracts." In: *2021 IEEE/ACM 4th International Workshop on Emerging Trends in Software Engineering for Blockchain*. IEEE, May 2021, pp. 17–26. DOI: `10.1109/WETSEB52558.2021.00010`.

[69] Sarwar Sayeed, Hector Marco-Gisbert, and Tom Caira. "Smart Contract: Attacks and Protections." In: *IEEE Access* 8 (2020), pp. 24416–24427. DOI: `10.1109/ACCESS.2020.2970495`.

[70] Fabian Schär. "Decentralized Finance: On Blockchain- and Smart Contract-based Financial Markets." In: *SSRN Electronic Journal* (2020). DOI: `10.2139/ssrn.3571335`.

[71] Fabian Schär. "Defi's Promise and Pitfalls." In: *International Monetary Fund* (Sept. 2022). URL: `https://www.imf.org/en/Publications/fandd/issues/2022/09/Defi-promise-and-pitfalls-Fabian-Schar` (visited on 01/10/2023).

[72] Jonas Schiffl et al. "Towards Correct Smart Contracts: A Case Study on Formal Verification of Access Control." In: *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies*. New York, NY, USA: ACM, June 2021, pp. 125–130. DOI: `10.1145/3450569.3463574`.

[73] Clara Schneidewind et al. "eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts." In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, Oct. 2020, pp. 621–640. DOI: `10.1145/3372297.3417250`.

[74] SmartCheck. *SmartCheck Solidity Overpowered Roles*. 2019. URL: `https://github.com/smartdec/smartcheck/blob/master/rule_descriptions/SOLIDITY_OVERPOWERED_ROLE/description_en.html` (visited on 05/18/2023).

[75] Solidity. *Solidity Cheatsheet*. URL: `https://docs.soliditylang.org/en/v0.8.18/cheatsheet.html` (visited on 04/27/2023).

[76] Mirko Staderini, Caterina Palli, and Andrea Bondavalli. "Classification of Ethereum Vulnerabilities and their Propagations." In: *2020 Second International Conference on Blockchain Computing and Applications*. IEEE, Nov. 2020, pp. 44–51. DOI: `10.1109/BCCA50787.2020.9274458`.

[77] The Coinbase Digital Asset and Protocol Security Team. *How Coinbase reviews tokens on Ethereum for technical security risks*. Oct. 2022. URL: `https://www.coinbase.com/blog/how-coinbase-reviews-tokens-on-ethereum-for-technical-security-risks` (visited on 12/06/2022).

[78] Sergei Tikhomirov et al. "SmartCheck." In: *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. New York, NY, USA: ACM, May 2018, pp. 9–16. DOI: `10.1145/3194113.3194115`.

[79] Roberto Tonelli et al. "Smart Contracts Software Metrics: a First Study." In: (Feb. 2018). arXiv: `1802.01517`.

[80] Christof Ferreira Torres, Hugo Jonker, and Radu State. "Elysium: Context-Aware Bytecode-Level Patching to Automatically Heal Vulnerable Smart Contracts." In: *25th International Symposium on Research in Attacks, Intrusions and Defenses*. New York, NY, USA: ACM, Oct. 2022, pp. 115–128. DOI: `10.1145/3545948.3545975`.

[81] Christof Ferreira Torres, Julian Schütte, and Radu State. "Osiris." In: *Proceedings of the 34th Annual Computer Security Applications Conference*. New York, NY, USA: ACM, Dec. 2018, pp. 664–676. DOI: `10.1145/3274694.3274737`.

[82] Christof Ferreira Torres et al. "ÆGIS: Shielding Vulnerable Smart Contracts Against Attacks." In: *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. New York, NY, USA: ACM, Oct. 2020, pp. 584–597. DOI: `10.1145/3320269.3384756`.

[83] Trail of Bits. *(Not So) Smart Contracts*. URL: `https://github.com/crytic/building-secure-contracts/tree/master/not-so-smart-contracts/algorand` (visited on 03/28/2023).

[84] Trail of Bits. *Slither Detector Documentation*. Mar. 2021. URL: `https://github.com/crytic/slither/wiki/Detector-Documentation/#calls-inside-a-loop` (visited on 05/18/2023).

[85] Petar Tsankov et al. "Securify." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, Oct. 2018, pp. 67–82. DOI: `10.1145/3243734.3243780`.

[86] Turgay Arda Usman, Ali Aydın Selçuk, and Süleyman Özarslan. "An Analysis of Ethereum Smart Contract Vulnerabilities." In: *2021 International Conference on Information Security and Cryptology*. IEEE, Dec. 2021, pp. 99–104. DOI: `10.1109/ISCTURKEY53027.2021.9654305`.

[87] Anqi Wang et al. "Artemis: An Improved Smart Contract Verification Tool for Vulnerability Detection." In: *2020 7th International Conference on Dependable Systems and Their Applications*. IEEE, Nov. 2020, pp. 173–181. DOI: `10.1109/DSA51864.2020.00031`.

[88] Huaimin Wang et al. "Blockchain Challenges and Opportunities: A Survey." In: *International Journal of Web and Grid Services* 14.4 (2018), p. 352. DOI: `10.1504/IJWGS.2018.10016848`.

[89] Wei Wang et al. “ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts.” In: *IEEE Transactions on Network Science and Engineering* 8.2 (Apr. 2021), pp. 1133–1144. DOI: `10.1109/TNSE.2020.2968505`.

[90] Xinming Wang et al. “ContractGuard: Defend Ethereum Smart Contracts with Embedded Intrusion Detection.” In: *IEEE Transactions on Services Computing* (2019), pp. 1–1. DOI: `10.1109/TSC.2019.2949561`.

[91] Ziling Wang, Qinyuan Zheng, and Ye Sun. “GVD-net: Graph embedding-based Machine Learning Model for Smart Contract Vulnerability Detection.” In: *2022 International Conference on Algorithms, Data Mining, and Information Technology*. IEEE, Sept. 2022, pp. 99–103. DOI: `10.1109/ADMIT57209.2022.00024`.

[92] Maximilian Wohrer and Uwe Zdun. “Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity.” In: *2018 International Workshop on Blockchain Oriented Software Engineering*. IEEE, Mar. 2018, pp. 2–8. DOI: `10.1109/IWBOSE.2018.8327565`.

[93] Zhendong Wu et al. “Detecting Vulnerabilities in Ethereum Smart Contracts with Deep Learning.” In: *2022 4th International Conference on Data Intelligence and Security*. IEEE, Aug. 2022, pp. 55–60. DOI: `10.1109/ICDIS55630.2022.00016`.

[94] Pengcheng Xia et al. “Trade or Trick? Detecting and Characterizing Scam Tokens on Uniswap Decentralized Exchange.” In: (Sept. 2021). arXiv: `2109.00229`.

[95] Junzhou Xu et al. “A Survey on Vulnerability Detection Tools of Smart Contract Bytecode.” In: *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education*. IEEE, Sept. 2020, pp. 94–98. DOI: `10.1109/ICISCAE51034.2020.9236931`.

[96] Yinxing Xue et al. “Cross-Contract Static Analysis for Detecting Practical Reentrancy Vulnerabilities in Smart Contracts.” In: *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, Dec. 2020, pp. 1029–1040. DOI: `10.1145/3324884.3416553`.

[97] Dylan Yaga et al. *Blockchain Technology Overview*. Tech. rep. Gaithersburg, MD: National Institute of Standards and Technology, Oct. 2018. arXiv: `1906.11078`.

[98] Qingren Zeng et al. “EtherGIS: A Vulnerability Detection Framework for Ethereum Smart Contracts Based on Graph Learning Features.” In: *2022 IEEE 46th Annual Computers, Software, and Applications Conference*. IEEE, June 2022, pp. 1742–1749. DOI: `10.1109/COMPSAC54236.2022.00277`.

[99] Jian Zhang et al. “A Novel Neural Source Code Representation Based on Abstract Syntax Tree.” In: *2019 IEEE/ACM 41st International Conference on Software Engineering*. IEEE, May 2019, pp. 783–794. DOI: `10.1109/ICSE.2019.00086`.

[100] Liyi Zhou et al. “SoK: Decentralized Finance (DeFi) Attacks.” In: (Aug. 2022). arXiv: `2208.13035`.

[101] Weiqin Zou et al. “Smart Contract Development: Challenges and Opportunities.” In: *IEEE Transactions on Software Engineering* 47.10 (Oct. 2021), pp. 2084–2106. DOI: `10.1109/TSE.2019.2942301`.

A. Overview of Vulnerabilities

Table A.1.: Smart contract vulnerabilities that might occur in Ethereum contracts

Name	**Description**
Denial of Service (DoS)	Occurs when a transaction is reverted due to a caller contract encountering a failure in an external call, or when the callee contract deliberately performs the revert operation to disrupt the execution of the caller contract [26, 2, 74, 76, 43].
Randomness using block data	Entropy sources are manipulable and should not be used as sources of randomness [51, 26, 53, 41, 23].
'tx.origin' Usage	Occurs when a contract uses tx.origin for authorization, which can be compromised by a phishing attack [51, 98, 50, 26, 2].
Integer overflow/underflow	Occurs when the result of an arithmetic operation falls outside of the range of a Solidity data type [51, 93, 41, 76, 43].
Re-entrancy	Occurs when an external callee contract calls back to a function in the caller contract before the caller contract finishes, allowing the attacker to bypass the due validity check until the caller contract is drained of Ether or the transaction runs out of gas [98, 50, 62, 42, 58].

Continued on the next page

Mishandled exception	If there is an exception raised in the callee contract, the callee contract terminates, reverts its state and returns false. However, depending on how the call is made, the exception in the callee contract may or may not get propagated to the caller. Oftentimes, exceptions are not handled properly [42, 55, 59, 51, 26].
(Dangerous) delegate call	This vulnerability is caused by the fact that a callee contract can update the caller contract's state variables [90, 42, 87, 59, 1].
Unprotected ether withdrawal	Occurs when a contract's funds can be withdrawn by any caller, who is neither the owner of the contract nor an investor who deposited funds to the contract [43, 23, 51, 76, 35].
Floating pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly [51].
Function's default visibility	Occurs when a function's visibility is incorrectly specified and thus permits unauthorized access [51, 2, 78, 90, 43].
Unprotected "self-destruct"	A contract can be killed by the contract's owner (or a trusted third-party) using the suicide or self-destruct method. When a contract is killed, its associated bytecode and storage are deleted. The vulnerability is caused by inadequate authentication enforced by a contract [51, 98, 50, 76, 35].
Ether lost in transfer	In a fund transfer, Solidity does not check the validity of the recipient's address. If funds are sent to an orphan address, Ethereum automatically registers for the address rather than terminating. Hence, no one can withdraw the transferred funds, which are effectively lost, since the address is not associated to any account. This can occur if the address is hard coded in a contract function [51, 26, 76].

Continued on the next page

Timestamp dependency	Occurs when a contract uses the block.timestamp as part of the triggering condition when executing a critical operation. Block data can be manipulated by a malicious actors [98, 89, 78, 42, 55].
Lack of transactional privacy	Since the blockchain is public, everyone can inspect the contents of a transaction, and infer the values of contract fields [51, 26, 76].
Transaction ordering dependency	The state of a contract may depend on how miners or validators select transactions to assemble into blocks [98, 89, 51, 50, 26].
Untrustworthy data feeds (oracles)	Oracles can supply deprecated or malicious content [26, 51].
Gas related issues	Gas related issues can cause a transaction to fail (e.g. out-of-gas exception) [50, 36].
Freezing ether	The ability of users to deposit their money to their contract accounts with the inability to spend their money from those accounts [50, 42, 76, 87, 59].
Permission verification missing	If the developer forgets to perform critical authorization checks on functions and if an attacker can execute arbitrary code [50, 53, 27, 91].
Type casts	Any address can be cast as a contract regardless of whether the code at the address represents the contract type being cast [26, 76, 43, 68, 23].
“send” instead of “transfer”	While “transfer” automatically checks for the return value, using “send” you have to manually check for the return value, and throw an exception if the send fails [26, 2, 78].

Continued on the next page

Redundant fallback function	Starting from Solidity 0.4.0, contracts without a fallback function automatically revert payments, making respective code redundant. The pattern detects the described construction (only if the pragma directive indicates the compiler version not lower than 0.4.0) [26, 2, 78].
Callstack depth	An attacker can recursively call a contract, at some point causing any subsequent external call made by the victim contract to fail [23].
Unsafe type inference	Explicitly define the type when declaring integer variables [2, 78].
Strict balance equality	Avoid checking for strict balance equality because an adversary can forcibly send ether to any account by mining or via selfdestruct [2, 78].
Outdated compiler version	Occurs when a contract uses an outdated compiler, which contains bugs and thus makes a compiled contract vulnerable [2, 78, 43, 23].
Arithmetic accuracy deviation	Solidity supports neither floating-point nor decimal types. For integer division, the quotient is rounded down [2, 78].
Unchecked low level call	When errors happen in low level functions in Solidity, a boolean value set to false is returned, but the code keeps running. Therefore, the result of such low level functions should be checked to confirm successful execution [41, 80].
Short address	If the length of the encoded arguments is shorter than expected, then EVM will auto-pad extra zeros to the arguments to make up for 32 bytes. This vulnerability is caused by EVM not checking the validity of addresses [53, 90, 69, 76].

Continued on the next page

Replay attack	Digital signatures can be used for identity authentication. However, by intercepting and replaying the user's previous signature, a malicious user can impersonate a specific user [76, 39].
Erroneous constructor name	If the constructor's name is misspelled, the intended constructor becomes a public, normal function that can be invoked by any address [43, 23].
Uninitialized storage pointers	For a compound local variable (e.g. struct, array, or mapping), a reference is assigned to an unoccupied slot in the storage to point to the state variable. If the local variable is not explicitly initialized, then the local variable's reference points to slot 0 by default, causing the content starting from slot 0 to be overwritten [23, 43].
Token API violation	Certain ERC20 functions (approve, transfer, transferFrom) return a bool indicating whether the operation succeded. It is not recommended to throw exceptions (revert, throw, require, assert) inside those functions [78, 2].
Upgradable contract	When the contract developer becomes malicious, the updated contract can be malicious [23].
Ownership takeover	A malicious actor can take over the ownership of a smart contract giving the attacker priveleged access rights which were not intended for that user initially [18].
Token Supply manipulation	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the customer [56].
Centralization of power	When single actors have the sole authority to execute a critical function [56].

Table A.2.: Smart contracts vulnerabilities that might occur in Algorand contracts

Name	**Description**
Frozen Token	The ability of users to deposit funds to a contract with the inability to withdraw the funds from those accounts, effectively freezing their funds [23, 13].
Insufficient signature information	This vulnerability causes a digital signature to be valid for multiple transactions [13].
Generating randomness	This vulnerability addresses smart contracts using PRNG to create random numbers [13].
Token lost to orphan address	There is a lack of validation checks on payment transactions regarding the recipient's address [23, 13].
Rekeying	The lack of check for RekeyTo field allows malicious actors to rekey the associated account and control the account assets directly, bypassing the restrictions imposed by the Teal contract [83].
Group Size Check	Lack of group size check in contracts that are supposed to be called in an atomic group transaction might allow attackers to misuse the application [83].
Access Controls	The vulnerability is caused by inadequate authentication enforced by a contract on critical contract functionality (eg. destroy contract) [83, 13].
Asset Id Check	Lack of verification of asset id in the contract allows attackers to transfer a different asset in place of the expected asset and mislead the application [83].

Continued on the next page

Denial of Service	Occurs if the execution of a transaction is reverted due to a thrown error or a malicious callee contract that deliberately interrupts the execution [83, 13].
Centalization related risks	High-privilege functions, which can only be invoked by certain group of accounts [21].