# Large-Scale Study of Vulnerability Scanners for Ethereum Smart Contracts

Christoph Sendner*, Lukas Petzi*, Jasper Stang*, Alexandra Dmitrienko*
*University of Würzburg, Germany

*Abstract*—**Ethereum smart contracts, which are autonomous decentralized applications on the blockchain that manage assets often exceeding millions of dollars, have become primary targets for cyberattacks. In 2023 alone, such vulnerabilities led to substantial financial losses exceeding a billion US dollars. To counter these threats, various tools have been developed by academic and commercial entities to detect and mitigate vulnerabilities in smart contracts. Our study investigates the gap between the effectiveness of existing security scanners and the vulnerabilities that still persist in practice. We compiled four distinct datasets for this analysis. The first dataset comprises 77,219 source codes extracted directly from the blockchain, while the second includes over 4 million bytecodes obtained from Ethereum Mainnet and testnets. The other two datasets consist of nearly 14,000 manually annotated smart contracts and 373 smart contracts verified through audits, providing a foundation for a rigorous ground truth analysis on bytecode and source code. Using the unlabeled datasets, we conducted a comprehensive quantitative evaluation of 18 vulnerability scanners, revealing considerable discrepancies in their findings. Our analysis of the ground truth datasets indicated poor performance across all the tools we tested. This study unveils the reasons for poor performance and underscores that the current state of the art for smart contract security falls short in effectively addressing open problems, highlighting that the challenge of effectively detecting vulnerabilities remains a significant and unresolved issue.**

## 1. Introduction

Blockchains are digital ledgers that enable mutually untrusted parties to transact without involving a third-party intermediary such as a bank. The rise and wide usage of blockchain platforms like Bitcoin [1], Ethereum [2], and Hyperledger [3] fueled rapid growth of the blockchain ecosystem.

Smart contracts, which are self-executing computer programs, are a crucial component of blockchain-based systems. They are deployed on a blockchain and facilitate the creation of new decentralized applications (dApp) [4], such as Decentralized Finance (DeFi) [5], Non-Fungible Tokens (NFTs) [6], and games [7]. Smart contracts govern the use of cryptocurrency – digital coins that are locked in a contract and can be accessed only when specific conditions are met. Developers can use them to build various services, for instance, for anonymizing money (e.g., TornadoCash [8]), lending money (e.g., MakerDAO [9]), and pooling resources for various projects (e.g., Uniswap [10]).

Decentralized applications have gained immense popularity, with MakerDAO alone holding $8.49 billion in cryptocurrency [11]. However, these dApps are often targeted by malicious actors who try to exploit vulnerabilities in their contracts. Unfortunately, some of these attacks are successful and can result in damages worth millions of dollars for all parties involved [12].

Cryptocurrency hacks have become increasingly common in recent years, with the DAO hack [12] being the first and one of the most well-known incidents. It resulted in a loss of $60 million US dollars and led to a hard fork of Ethereum. Other examples include the Safemoon Hack, which occurred due to an access control vulnerability and enabled adversaries to exfiltrate around 8.9 million dollars [13]. Furthermore, the LendHub hack took place when an attacker exploited a wrong update mechanism to steal approximately 6 million dollars [14]. In the Deus Finance hack, an attacker exploited an access control issue to steal 13.4 million dollars [15].

Smart contract developers face challenges in dealing with vulnerabilities and bugs, as the most traditional approach of code patching is not applicable due to the immutability of the underlying blockchain. They have to resort to smart contract update mechanisms, which, as we can see from the LendHub example, can themselves have errors and lead to exploitable vulnerabilities. Alternatively and preferably, smart contracts have to be subjected to rigorous code analysis in pre-deployment phase, before the code is uploaded to the blockchain and becomes immutable.

In recent years, many vulnerability detection tools have been developed for smart contracts that can aid smart contract developers in pre-deployment security testing. These tools can be classified into four categories: Symbolic analysis [16]–[19], static analysis [20], [21], machine learning approaches [22]–[24], and fuzzing [25]–[27]. Interestingly, even though some of these tools have been available for years and provided under open-source licenses, smart contracts continue to suffer from vulnerabilities that malicious actors can exploit. In 2023 alone, hackers managed to exploit vulnerabilities in smart contracts, resulting in gains exceeding one billion US dollars [28].

In this work, we aim to shed light on the problem of smart contract security testing. In particular, we want to identify the cause of the problem and answer the following questions: *Why, despite the existence of many effective vulnerability detection tools, does the problem of vulnerabilities in smart contracts still prevail? Is it due to difficulties of setting up and using those tools, which raises the adaption barrier, or because they are less effective than the security* *research community believes? If some tools are better suitable for the detection of selected vulnerability types, can one achieve better detection by using several tools?*

To answer these questions, we want to study existing vulnerability detection tools with the goal of accessing and comparing their effectiveness. Overall, we aim to understand if the research field devoted to vulnerability detection in smart contracts is sufficiently researched or if there are remaining open problems.

We acknowledge the efforts of previous works that have attempted already to compare various tools for vulnerability detection [29]–[42]. However, these works either provide a survey-like comparison without performing actual performance evaluations [29]–[34] or evaluate a small subset of tools on a very limited dataset [35], [38]–[42]. In addition, some of these works [36], [38], [39], [41] only analyze tools that operate either on bytecode or source code and do not offer a comprehensive exploration of both. Furthermore, previous studies have not explored the potential of bundling different tools to enhance vulnerability detection.

In our research, we seek to understand why vulnerabilities continue to be exploited despite the existence of various scanning tools. In particular, we aim to answer the above-postulated questions by means of conducting a comprehensive study of the existing tools. This study is to be performed using a range of publicly accessible resources and extensive datasets, including bytecode, source code, and ground-truth data.

**Contributions.** We make the following contributions:

- We conduct a large study of existing vulnerability detection tools by building, using, and comparing 18 smart contract vulnerability scanners from different methodology categories that utilize static analysis, symbolic execution, fuzzing, and machine learning techniques for detection. Among them, 14 tools perform source code-based analysis, while four detect vulnerabilities at the bytecode level. Three of the tools allow for the analysis of both bytecode and source code. Related literature considered only a subset of tools [35], [39], [41], [42] or discussed only a particular methodology [38], [40]. Unlike previous large-scale analysis [39], our study integrates an analysis of bytecode, conducts a comprehensive analysis of the employed tools, and is not limited to just reentrancy bugs but considers eight distinct vulnerabilities. Furthermore, our research is underpinned by a more extensive dataset and a broader range of tools, setting a new benchmark in the field of smart contract vulnerability analysis.
- To conduct our study using tools that expect source code as input, we built a dataset comprising 77,219 unique and carefully de-duplicated smart contracts. We collected the source codes from EtherScan [43] and InterPlanetary File System (IPFS) [44], where some developers upload their source code. This is the largest dataset of real-world source codes directly from Ethereum blockchain available to date.
- To study tools operating on bytecode of smart contracts, we built an unprecedentedly large dataset of 4,062,844 bytecodes after de-duplication of initially collected 26,740,370 bytecodes. For this dataset, we tried to get as many Ethereum-compatible smart contracts as one could find. In particular, we collected all smart contracts from Ethereum network and extended it with smart contracts collected from four test networks: Goerli [45], Rinkeby [46], Ropsten [47], and Kovan [48].
- We performed a comprehensive study of 14 source code-based tools (Slither [20], SmartCheck [21], Maian [16], Oyente [17], Artemis [49], Osiris [50], Securify2 [19], Mythril [51], TeEther [18], ConFuzzius [25], Smartian [27], sFuzz [26], GNNSCVulDetector [22], MANDO-GURU [52]) and attempted to detect 8 vulnerability types in our source code-based dataset. The goal of this study was to establish if the outcomes of these tools are consistent with each other, and if one could potentially use a strategy of using several detection tools to enhance detection performance. The outcomes of this study are non-trivial – we observed significant discrepancies in detection outcomes. For instance, we reveal that the tools do not agree on a single sample to have the reentrancy vulnerability. This shows that the idea of combining several tools won't be very useful in practice.
- We additionally studied four vulnerability detection tools that operate on the byte-code level: Vandal [53], Maian [16], Oyente [17], Mythril [51]. Their performance was evaluated using 5 vulnerability types. The results of this evaluation similarly display lack of consensus among the tools, to the extent that questions their ability for accurate detection.
- Significant discrepancies in detection outcomes of both source code and bytecode-based tools indicate that at least some of them do not provide good detection performance. To verify this preposition, we built two additional datasets labeled with ground-truth labels. Our team manually labeled one ground truth dataset for the reentrancy bug, consisting of 13,773 unique smart contracts. The other dataset comprises 373 unique smart contracts collected from publicly available repositories and includes eight vulnerability types. Vulnerabilities in these contracts were confirmed by security audits conducted by security firms.
- Equipped with the two ground truth datasets, we evaluate the performance of both source code and bytecode-based tools involved in our study. Our results highlight the overall poor performance of all tools under our evaluation, with F1-scores ranging from 0% to a maximum of 73%. This is primarily attributed to the high incidence of false positives and negatives reported by the tools.
- The results of our analysis reveal the reasons why the performance of the tools under evaluation is often below expectations and derive other valuable insights. We provide insights into these reasons. For instance, we observe that varying compiler versions can lead to significant changes that obstruct the effectiveness of

vulnerability scanners in their analysis. Additionally, the evolving coding practices of developers could also affect the performance of these scanners. Our findings aim to assist future research in developing more efficient tools for detecting vulnerabilities.
- We plan to make datasets and tools available for further research at https://github.com/sss-wue/sc-study/.

Overall, our study demonstrates that the current state of the art in the area of smart contract security has significant room for improvement and the problem of vulnerability detection remains an open and challenging problem.

**Outline.** The rest of this paper is structured as follows: We offer in Section 2 a brief overview of smart contract vulnerabilities and the tools we utilized in this study to detect them. In Section 3, we describe our datasets and the methodology employed in their construction. The analysis of the different tools is presented in Section 4. An additional discussion around security analysis of smart contracts is provided in Section 5. We examine related work in Section 6. We conclude the paper in Section 7.

## 2. Background

This section provides the background information for smart contracts, their vulnerabilities, and the vulnerability scanners. We will especially focus on those vulnerabilities and scanners that will be further explored in this study.

### 2.1. Smart Contracts

A smart contract refers to a software program that operates within a blockchain environment. In this study, we focus on smart contracts designed for the Ethereum blockchain. These contracts are typically written in Solidity [54], compiled, and executed within the Ethereum Virtual Machine (EVM). The EVM functions as a stack-based machine with a word size of 256 bits and a stack size of 1024, utilizing a word-addressable memory model [55].

To deploy a smart contract, the compiled EVM bytecode is uploaded to the blockchain through a transaction. Interacting with the smart contract is also achieved by sending transactions to the contract's address. It is important to note that executing smart contracts incurs a cost in the form of Gas, which is essentially a fee that aims to cover the costs of execution (e.g., electricity costs). Certain contracts incorporate an IPFS link within their bytecode by encoding essential metadata. This link grants access to retrieve valuable information, including the Application Binary Interface (ABI), source code, and additional metadata.

### 2.2. Smart Contract Vulnerabilities

Similar to any other software, smart contracts are susceptible to bugs and vulnerabilities. Given that smart contracts are directly associated with cryptocurrencies, the potential financial losses resulting from undiscovered vulnerabilities can be significant. Numerous vulnerabilities exist in the realm of Solidity smart contracts and are listed in the Smart Contract Weakness Classification Registry (SWC) [56]. One can classify these vulnerabilities into three categories: Software Errors, Runtime Bugs, and Blockchain Characteristics.

**Software Errors** In the field of Solidity smart contracts, software errors often result in bugs, with arithmetic issues (SWC-101 [56]) such as integer overflows and underflows being particularly prevalent. Notably, these vulnerabilities have been partly addressed through a compiler update in Solidity. However, for comprehensive legacy detection, these issues remain a focus of this study.

Additionally, various other critical bugs warrant attention. The 'suicide' vulnerability (SWC-106 [56]), for example, can potentially allow an attacker to destroy a contract if its functions lack proper safeguards. Equally concerning is the assert violation [57], which arises when developers inadvertently leave a failing assert statement in live code. Also of significance is the misuse of txOrigin (SWC-115 [56]), which, if exploited in place of msg.sender, could allow attackers to manipulate a global variable to their advantage.

Another notable vulnerability involves the use of block timestamps as a proxy for time, which can be exploited by attackers aware of this time dependency (SWC-116 [56]). Lastly, the issue of 'Locked Ether' arises from the absence of a mechanism to withdraw Ether from the contract, leading to potential fund loss if not addressed pre-deployment.

**Runtime Bugs** Smart contracts also face a range of bugs associated with their execution runtime. These vulnerabilities arise due to specific characteristics of how smart contracts are executed. For instance, an example vulnerability is reentrancy (SWC-107 [56]), where an attacker can exploit the ability to reenter a function during runtime, even before the initial execution is completed. Another example is the legacy callstack depth vulnerability [58], where the stack can become exhausted. Additionally, there are "greedy contracts" [16], which only accept Ether without providing a means for later extraction. Another critical vulnerability is associated with the use of DelegateCall (SWC-112 [56]). This feature allows a contract to execute another contract's code within its own context. It's essential to emphasize that the external contract's code must be thoroughly vetted for security, as any compromise in its integrity could give an attacker complete control over the caller's funds.

**Blockchain Characteristics** The blockchain infrastructure itself can give rise to vulnerabilities within smart contracts. An instance of such vulnerability is money concurrency or Transaction Ordering Dependency (ToD) (SWC-114 [56]), which can result in financial losses when the code depends on the sequence of transactions, such as determining the first correct answer submitted [59]. Another concern emerges when developers utilize blockchain's block values for time-related purposes. However, these values can be influenced by miners and should not be relied upon. Additionally, creating randomness within smart contracts becomes challenging due to their deterministic execution nature, potentially leading to vulnerabilities.

## 2.3. Vulnerability Scanners

In the following, we categorize each vulnerability scanner into one of four analysis approaches: Static Analysis, Symbolic Execution, Fuzzing, and Machine Learning.

**Static Analysis** Static Analysis involves examining source code or compiled output without execution. This allows tools to detect bugs and security issues without requiring an execution environment or risking running vulnerable or malicious code.

*Slither* [20] is a static analysis framework for finding vulnerabilities in Solidity source files. It supports the detection of over 80 vulnerability classes and can be extended. Slither supports the detection of, for example, suicidal contracts, reentrancy, and block data dependency.

*Vandal* [53] is a security analysis framework designed for EVM bytecodes. It operates by converting the bytecode into semantic logic relations, which are subsequently analyzed against specified vulnerabilities using a declarative language. The tool detects a range of vulnerabilities, including Unchecked Send, Reentrancy, and Selfdestruct.

*SmartCheck* [21] conducts thorough lexical and syntactical analyses on source code to identify vulnerabilities. It employs text parsing techniques to convert Solidity source code into an XML parse tree, which is subsequently examined for vulnerability patterns. It is crucial to emphasize that as of 2020, SmartCheck has been deprecated and is no longer actively maintained.

*EtherTrust* [60] classifies bytecodes by defining an abstract EVM semantic representation and rules for detecting reentrancy vulnerabilities. It abstracts EVM bytecode and employs static reachability analysis with Horn clauses for vulnerability detection.

**Symbolic Execution** Differing from Static Analysis, Symbolic Execution encompasses the execution of the specific source code or compiled code, including dynamic analysis. Given the overlapping nature of symbolic execution and dynamic analysis, these tools are categorized together under this unified category.

*Maian* [16] is a symbolic analysis tool for EVM bytecodes and Solidity source codes. It detects three types of vulnerabilities: Prodigal, Suicidal, and Greedy contracts.

*Oyente* [17] is a symbolic execution tool created for detecting possible security flaws in the source code and bytecode of Ethereum smart contracts. Like SmartCheck, Oyente is also outdated and no longer actively maintained.

*Artemis* [49] and *Osiris* [50] are extensions of Oyente that aim to detect additional vulnerabilities.

*Securify2* [61] is an analysis tool based on Securify [19] that was deprecated since 2019. Similar to the initial version, Securify2 performs in-depth analysis of EVM bytecode, examining it against a predefined set of security patterns. These patterns consist of both compliance and violation patterns, which encompass the necessary conditions for ensuring the smart contract's adherence to specific security requirements or, conversely, flagging instances where predefined security properties are violated.

*EthBMC* [62] is a bounded model checker analysis tool that utilizes symbolic execution to examine smart contract EVM bytecodes against predefined models. It specializes in detecting suicidal contracts and contracts that permit arbitrary extraction of funds.

*Mythril* [51] is a EVM bytecode analysis tool. The tool combines symbolic execution, SMT solving, and taint analysis techniques to detect a wide range of vulnerabilities, such as suicidal contracts or contracts that rely on weak sources of randomness.

*TeEther* [18] does not primarily focus on general vulnerability detection but rather the creation of practical exploits. To achieve this, it specifically targets four low-level instructions closely tied to value transfer: (1) Call, (2) Selfdestruct, (3) Callcode, and (4) Delegatecall. By imposing path constraints and evaluating their satisfiability, the tool examines each execution path to determine if an undesired value transfer could potentially occur.

**Fuzzing** Although fuzzing can be categorized as a form of runtime analysis, we distinguish it from other techniques based on the selection of input data. Fuzzing involves the guided but random selection of input data, setting it apart from other methods in the Symbolic Execution category.

*ILF* [63] is a fuzzing tool that classifies solidity source code files. The fuzzer is trained from data generated by a symbolic execution engine. The trained fuzzing model can be used to detect vulnerabilities such as Suicidal contracts, Leaking vulnerability or Block dependency.

*ConFuzzius* [25] is an evolutionary fuzzer that applies constraint resolving and data dependency analysis to detect smart contract vulnerabilities. The fuzzer detects 10 vulnerabilities, including Suicidal, Reentrancy, and ToD.

*Smartian* [27] is a mutation-based fuzzer that integrates static and dynamic analyses to drive input mutation. By leveraging dynamic dataflow analysis, Smartian dynamically guides the fuzzing engine and incorporates bug oracles into its testing process.

*sFuzz* [26] is a framework developed based on AFL [64], a renowned fuzzer primarily used for C/C++ programs. It extends the capabilities of AFL to support EVM bytecode by introducing adaptations in multiple areas. Notably, sFuzz introduces a novel metric to target hard-to-cover branches specifically.

**Machine Learning** Machine Learning has been widely utilized in various domains to identify vulnerabilities, and the realm of smart contracts is no exception. In this context, we introduce two available tools that leverage Graph Neural Networks (GNNs) and Recurrent Neural Networks (RNNs), respectively, for vulnerability detection purposes.

*GNNSCVulDetector* [65] is designed to detect vulnerabilities in smart contracts by utilizing a GNN. It analyzes the syntactic and semantic structures of the smart contract, which are represented as a graph. This graph is processed using a degree-free graph convolutional neural network for classification. The tool identifies three vulnerability classes: Reentrancy, Timestamp Dependency, and Infinite Loop. While the source code for the generation of the datasets and training the model is publicly available, execution is restricted. With minor bug fixes the dataset generation and training was successful for the timestamp vulnerability but not for Reentrancy and Infinite Loop vulnerabilities.

*MANDO-GURU* [52] utilizes a topology GNN to derive node embeddings from a heterogeneous code graph, followed by graph-level vulnerability classification. Should the code graph be deemed vulnerable, a node classification model is trained to pinpoint the vulnerable nodes. Our focus is on the graph-level labels (determining if the source code has vulnerabilities) rather than the finer-grained node-level labels (identifying the specific locations of vulnerabilities).

# 3. Datasets

Ethereum contracts are stored across various blockchains, primarily encompassing the Ethereum Mainnet (i.e., the primary production blockchain) and the testnets, namely Goerli [45], Rinkeby [46], Ropsten [47], and Kovan [48]. We successfully generated four distinct datasets from those blockchains. The first dataset is the Source Code Dataset (SCD), which comprises a collection of smart contract source codes. The second dataset is the Bytecode Dataset (BCD), consisting of a hexadecimal representation of compiled bytecodes of the smart contracts. The third dataset is the Reentrancy Ground Truth (RGT), specifically focused on identifying and labeling source code operations that are vulnerable to reentrancy. Lastly, the fourth dataset is the Audits Ground Truth dataset (AGT), which includes labeled data obtained from security audits conducted on smart contracts.

## 3.1. Source Code Dataset

The process of assembling our source code dataset involved two distinct stages. In the initial phase, we used Google BigQuery [66] to gather addresses of smart contracts. The second phase entailed downloading the source code, which was accessible either through EtherScan [43] or IPFS [44].

EtherScan [43], a prominent blockchain explorer and analytics tool for the Ethereum network, offers a detailed set of functionalities for examining and extracting information about Ethereum transactions, addresses, smart contracts, and overall network dynamics. One of its key features is providing access to the source code of specific contracts on the Mainnet, enabling a deeper understanding of the contracts' operations and verifying their functionalities. We utilized the EtherScan API to systematically download all accessible source codes from the Mainnet blockchain.

Regarding IPFS, our approach started with downloading the contract bytecodes to check for embedded metadata. This metadata, located at the end of the bytecode, was then decoded using a CBOR decoder. From the decoded information, we extracted the IPFS link, which led us to the original contract metadata containing the source code.

After implementing a deduplication process by hashing the smart contracts and comparing these hashes, we successfully compiled a dataset of 77,219 unique source codes, all written in the Solidity language.

## 3.2. Bytecode Dataset

We retrieve the bytecode from Ethereum contracts on different blockchains.

To synchronize with the Ethereum blockchain, we employed two clients: Erigon [67] for the Goerli, Ropsten, and Rinkeby testnets, chosen for its speed and lower disk space usage, and Geth [68] for the Kovan network, as it is not yet supported by Erigon. Furthermore, Ethereum Mainnet contracts were sourced from an open dataset available through Google's BigQuery service [66]."

We proceeded by retrieving the contract addresses from each blockchain and utilizing the Python Web3 API library [69] to extract the EVM bytecode from the downloaded blockchains. These extracted bytecodes were then stored in a MySQL database. Some contracts obtained this way were found to be empty. There are a few possible reasons for this. It could be due to the Ethereum node not being fully synchronized with the network, resulting in an unavailable bytecode. Alternatively, it could be because empty contracts were deployed on the blockchain or the contract had been self-destructed. Overall, we successfully extracted 26,740,370 non-empty bytecodes from the five networks. Specifically, the Ethereum Mainnet contributed 22,789,100 bytecodes, Ropsten provided 1,831,168, Rinkeby contributed 1,382,338, while Kovan and Goerli supplied 635,766 and 101,998 bytecodes, respectively. After deduplication, we can use 4,062,844 unique bytecodes for our analysis.

## 3.3. Reentrancy Ground Truth

This dataset is centered explicitly on reentrancy vulnerabilities, selected due to their frequent detection by smart contract vulnerability scanners. This focus allows us to compare across multiple tools. Given the intensive nature of manually labeling numerous contracts, our study is concentrated exclusively on this particular vulnerability. Consequently, we have developed a specialized source code dataset enriched with detailed annotations to deepen our investigation into reentrancy vulnerabilities. As a foundation, we utilized the SmartBugs Wild Dataset [70], which encompasses 47,398 source codes from various smart contracts. We began by eliminating duplicates from the dataset and then proceeded to incorporate the annotations specific to the reentrancy vulnerability. This dataset allows for a more detailed analysis of how effectively reentrancy vulnerabilities are detected in smart contract source codes and bytecodes downloaded directly from the blockchain.

**Preprocessing** During the preprocessing step, our focus was on removing duplicates from the dataset. These duplicates often arise when source code is copied and subsequently subjected to minor modifications, such as comment adjustments, variable or function renaming, or changes to variable values. To identify such contracts, we utilized the Solidity compiler to generate an Abstract Syntax Tree (AST) from the source code file. This AST representation allowed us to eliminate comments and whitespace characters while also removing intermediate values, variable names, and function names. This process facilitated effective comparison between Solidity files. By assessing the similarity of the AST trees by comparing their hashes, we identified contracts that displayed little resemblance to others, resulting in a refined dataset comprising of 22,237 source code files.

**Annotation** Subsequently, we proceeded to annotate the deduplicated dataset. The original SmartBugs Wild dataset only provided annotations for the reentrancy vulnerability for the call subtype. To achieve a more comprehensive and detailed annotation, we expanded our analysis to include all three subtypes: call, send, and transfer. The 'call' subtype is particularly critical, as its function lacks intrinsic gas limitations. On the other hand, 'send' and 'transfer' subtypes are designed to use only a specified amount of gas for the contract call, as defined in the Ethereum Yellow Paper [55]. However, this amount may vary with updates to the Ethereum network. It is important to note that contracts lacking these subtypes were considered non-vulnerable.

For each contract, we manually inspected the source code containing the three subtype functions. Our assessment focused on determining whether a state change occurred after the transfer of funds and whether a reentrancy occurred. Upon meeting these conditions, we annotated the respective contract as vulnerable to reentrancy attacks. As a result, our final reentrancy dataset comprises 13,773 smart contracts. Labeling this dataset took over four person-months by two master students, supervised by a Ph.D. student in IT security. Due to high efforts required to label the entire dataset, we stopped labeling after processing 13,773 instances.

### 3.4. Audits Ground Truth

Obtaining a ground truth dataset for evaluating the detection performance of various tools can be challenging and time-consuming. It requires significant investment in correctly labeling the data, often involving the expertise of human auditors. This process, known as security audits, ensures that the presence of vulnerabilities in smart contract source code is accurately identified. To acquire a reliable ground truth dataset for tools that operate at the level of the source code, we utilized publicly available audit repositories from reputable sources, including Quantstamp [71], OpenZeppelin [72], Trail of Bits [73], ConsenSys [74], and CertiK [75]. The resulting dataset consists of 373 smart contract source codes that have been labeled by security auditors into the vulnerability categories. This carefully labeled ground truth dataset provides a robust benchmark for evaluating the performance of different tools in vulnerability detection. Further, we compiled the available source codes to allow a bytecode-based analysis.

## 4. Study

In this section, we detail our study conducted with 13 source code-based and four bytecode-based tools, using the four datasets described in Section 3. The SCD and BCD datasets enable quantitative evaluation, while the RGT and AGT datasets facilitate qualitative analysis of the tools.

### 4.1. Methodology

In this section, we will discuss our methodology for the analysis of source codes, bytecode, and describe our visualization method.

We use the vulnerability scanners as-is and do not optimize them per smart contract. For instance, if the tool supports only a limited range of compiler versions, we don't attempt to enhance it to other versions. We also consider vulnerability scanners that time out on a smart contract as non-vulnerable since the outcome is the same for a developer – no vulnerability is found or reported. For a detailed evaluation of scanning robustness, we refer the reader to Section 4.7, where we analyze the completion rate of the different scanners.

**Source code analysis** We employed 13 tools to identify over 200 vulnerability types. For comparability and simplicity, our focus was on eight types detectable by at least three scanners. Results for other types are omitted due to space limitations.

**Bytecode analysis** We analyzed the bytecode-based datasets using four tools. Since the tools' density is lower than that of those operating on source codes, we provided our analysis of tools based on four vulnerability types present across the tools.

**Visualization** We opted for Upset plots that proved to be superior to Venn diagrams for visualizing complex intersections in datasets, as they provide a clearer and more scalable representation of the relationships between multiple sets, especially when dealing with large numbers of sets where Venn diagrams become cluttered and less interpretable.

Each plot comprises two main sections: On the left side, we display the total number of vulnerable samples found by each tool. On the top, we show the total number of samples that overlap among the tools. If there is a single dot, it signifies that these samples do not overlap with any other tool. Conversely, when a column contains multiple dots, it signifies that the respective tools agree in their analysis of these specific samples.

**Vulnerabilities** We focus on eight types of vulnerability throughout this paper: Suicide, Reentrancy, Transaction Order Dependency (ToD), Arithmetic Bugs, Usage of txOrigin, Time Dependency, Locked Ether, and DelegateCall.

**Test Environment** All tests were conducted in our High-Performance Cluster, in which each node comprises two Intel® Xeon Gold 6134 Processors (8c/16t), with 384 GB DDR4 memory and BeeGFS [76] for storage. We use

Docker [77] to parallelize the use of different tools and, thus, maximize the usage of available resources in the cluster.

**Effort** Deploying and extensively testing the tools on the datasets required considerable manual labor, totaling twelve person-months. Setting up the tools demanded between 600 to 800 hours, while over 1,000 hours were dedicated to gathering and labeling the datasets, followed by several months of computational time.

**Timeouts** We apply the following timeouts during evaluation to make a large-scale analysis possible: For Slither, Maian, Oyente, Artemis, and Mythril, we applied a timeout of 5 minutes. TeEther, Osiris, Smartian, ConFuzzius, and sFuzz had a timeout of 2 minutes. GNNSCVulDetector, MANDO-GURU, Securify2, and SmartCheck offer no possibility of defining a timeout, but their analysis is finished within a reasonable time frame.

### 4.2. Quantitative Analysis on SCD Dataset

As per our analysis, we have evaluated all 77,219 smart contracts available in SCD using 13 source code-based tools, listed in Table 1. We utilized them to detect eight types of vulnerabilities, namely: *Suicide*, *Reentrancy*, *Transaction Order Dependency (ToD)*, *Arithmetic Bugs*, *Usage of txOrigin*, *Time Dependency*, *Locked Ether*, and *DelegateCall*. During our analysis, we found that some tools provide a more detailed analysis of the reentrancy vulnerability type by reporting vulnerability sub-types. For example, the reentrancy vulnerability type can be divided into *bad*, *No Eth*, or *Benign*, as per the analysis by Slither and Securify2. Additionally, vulnerabilities like *Arithmetic Bugs* have been grouped based on the type of bugs, such as *Integer Overflow/Underflow* and generic *Arithmetic Bugs*.

**Suicide** We utilized seven distinct tools to identify occurrences of suicidal contracts in the source code. These tools are Slither, Mythril, Smartian, Confuzzius, Securify2, Maian, and TeEther. In Figure 1, we have visually presented the results obtained from running these tools. The figure clearly illustrates that none of the contracts were identified as vulnerable by all seven tools. The highest level of agreement was six out of seven tools, and this only occurred for three smart contracts, as indicated in the figure's last two columns. Summing up the columns that contain at least three dots reveals that a mere 70 contracts were marked as vulnerable by three or more tools.

The overlap between the scanners is minimal, with four tools independently detecting the suicide vulnerability in over 50 % of all tool-flagged contracts. Slither and TeEther have the most overlap with other tools in this analysis. However, even in these cases, the overlap with other tools is still limited.

**Reentrancy** Our study shows that there is minimal overlap between different tools when it comes to detecting the reentrancy vulnerability in smart contracts, as illustrated in Figure 2. We could utilize ten different tools to detect this vulnerability, thanks to the popularity of this vulnerability type among various scanners. Although Slither and MANDO-GURU stand out from the rest with an extremely high positive rate, including the positives of other tools and each other (MANDO-GURU uses Slither to construct its graphs), the most significant finding is that no single contract is marked as vulnerable when all the tools' results are combined. Out of the total of 77,219 smart contracts checked, only 106 were identified as vulnerable by three tools at most. This highlights the need for a comparative study of various tools to comprehensively and accurately assess this smart contract vulnerability.

**ToD** In our evaluation of the ToD vulnerability, as shown in Figure 3, none of the smart contracts were identified as vulnerable by all five tested tools. Additionally, not even four out of five tools agreed on a single contract. However, there was a significant overlap between Osiris and Artemis, which is not surprising since they are both based on the same underlying tool, Oyente. Oyente also overlaps with Artemis and Osiris, although it provides the fewest positive detections.

**Arithmetic Bugs** The evaluation results for reported arithmetic bugs are presented in Figure 4, which displays the test results for seven underlying tools. Remarkably, Artemis is notably absent from the figure, as it did not detect a single bug. Similar to previously observed trends, the tools agreed on no cases. It is noteworthy that Oyente, Osiris, and Confuzzius showed a significant overlap and agreement on an arithmetic bug in 2,519 samples. While the similarity of Osiris and Oyente can be explained by the fact that Osiris is based on Oyente, the reasons for the overlap between Confuzzius and Oyente are not that apparent. Generally, the overlap between the tools is more pronounced than with the other vulnerabilities. We attribute this result to the simplistic nature of this vulnerability type. But we also note that this positive result is undermined by the fact integer over- and underflow vulnerability is mitigated directly by the compiler.

**txOrigin** Based on our evaluation results, Figure 5 indicates that all five analyzed tools agree on only one sample to have the vulnerability. We observed that Slither overlaps with SmartCheck for most of its detected samples, which is quite interesting. Other than that, the tools mostly disagree, similar to the other vulnerabilities.

**Time Dependency** We use ten security scanners to identify the Time Dependency vulnerability. The Figure 14 from the Section A shows our evaluation results where not a single contract was identified as vulnerable by all tools. Eight tools agree on the existence of the vulnerability in seven samples out of our dataset. Similar to the reentrancy vulnerability, Slither identifies the most potentially vulnerable samples. Therefore, most of the other tools have a significant overlap with Slither. However, the majority of positive samples are disjoint.

**Locked Ether** According to the evaluation results presented in Figure 6, none of the six tools agree on a single sample. SmartCheck flags the most contracts as vulnerable and has a significant overlap with Slither. Securify2 has most of its flagged contracts overlapping with Slither and SmartCheck. Maian, on the other hand, only overlaps with the other tools on 20 samples.

| Metrics | Static Analysis | | Symbolic Execution | | | | | | | Fuzzing | | | Machine Learning | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Slither | SmartCheck | Maian | Oyente | Artemis | Osiris | Securify2 | Mythril | TeEther | ConFuzzius | Smartian | sFuzz | GNNSCVulDetector | Mando-GURU |
| Suicidal | ● | ○ | ● | ○ | ○ | ○ | ● | ● | ● | ● | ● | ○ | ○ | ○ |
| Reentrancy | ◒ | ○ | ○ | ● | ● | ● | ◒ | ● | ○ | ● | ● | ● | ○ | ● |
| ToD | ○ | ○ | ○ | ● | ● | ● | ◒ | ○ | ○ | ● | ○ | ○ | ○ | ○ |
| Arithmetic Bug | ○ | ○ | ○ | ◒ | ◒ | ◒ | ○ | ● | ○ | ◒ | ● | ○ | ○ | ● |
| TxOrigin | ● | ● | ○ | ○ | ● | ○ | ● | ● | ○ | ○ | ● | ○ | ○ | ○ |
| Timestamp | ● | ○ | ○ | ● | ● | ● | ● | ● | ○ | ● | ● | ● | ● | ○ |
| Lock | ● | ● | ● | ○ | ○ | ○ | ● | ○ | ○ | ● | ○ | ● | ○ | ○ |
| DelegateCall | ◒ | ○ | ○ | ○ | ● | ○ | ● | ● | ● | ● | ○ | ● | ○ | ○ |

TABLE 1: Overview of vulnerability scanners with detectable vulnerability types on source code. (●: Tool detects vulnerability, ◒: Tool detects finer granularity, ○: Tool does not detect vulnerability)

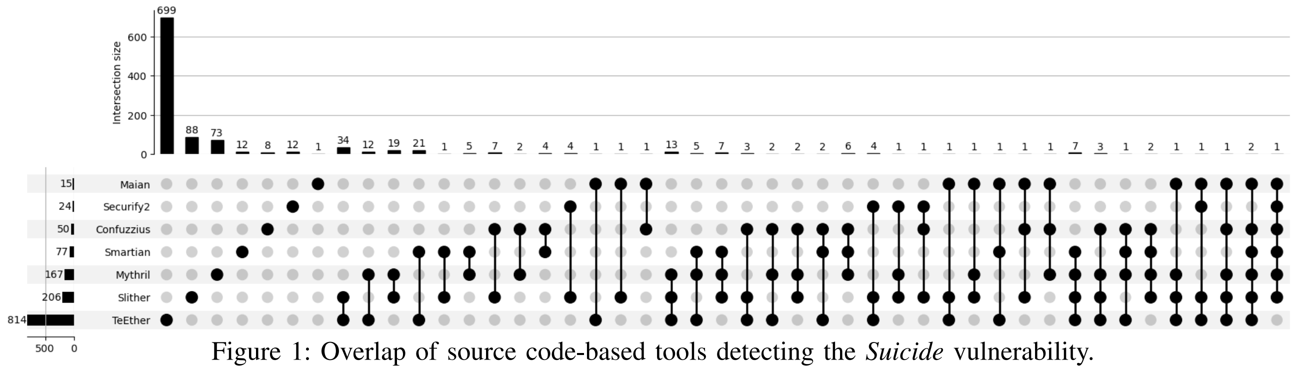


Figure 1: Overlap of source code-based tools detecting the *Suicide* vulnerability.

**DelegateCall** The Figure 7 shows our evaluation results of six tools. As with the other vulnerability types, none of the tools agree on a single example. Further, there is a substantial overlap between Mythril and other tools – specifically TeEther and Artemis.

**Summary** In summary, our quantitative analysis of all detectable vulnerabilities demonstrates a general lack of consensus among tools regarding the presence of any vulnerability. This indicates a significant gap between the reported detection capabilities and the actual data observed on-chain. The divergence in our detection results raises questions about the reliability of these tools in accurately identifying vulnerabilities in source code.

## 4.3. Quantitative Analysis on BCD Dataset

| Metrics | Static Analysis | Symbolic Execution | | |
|---|---|---|---|---|
| | Vandal | Maian | Oyente | Mythril |
| Suicidal | ● | ● | ○ | ● |
| Reentrancy | ● | ○ | ● | ● |
| UncheckedCall | ● | ○ | ○ | ● |
| txOrigin | ● | ○ | ○ | ● |
| Timestamp | ○ | ○ | ● | ● |

TABLE 2: Overview of vulnerability scanners with detectable vulnerability types on bytecode.

In this section, we evaluate four tools that can detect vulnerabilities in bytecode against our Bytecode dataset of 4,062,844 unique bytecodes of BCD (cf. Section 3.2). We compare each tool's performance, detecting five different vulnerabilities: Reentrancy, Suicidal, Unchecked Call, Use of txOrigin, and Time Dependency. We list an overview of the evaluated tools and their overlap for the detected vulnerability types in Table 2.

**Reentrancy** As highlighted, a reentrancy vulnerability permits an attacker to deplete a smart contract's funds. Figure 8 illustrates the intersections in detecting the Reentrancy bug among Mythril, Oyente, and Vandal. Vandal identifies the highest number of contracts vulnerable to reentrancy, leading to its greatest overlap with the other two tools. Conversely, Oyente and Mythril coincide in their identification on merely five contracts, while an overlap among all three tools occurs in just six distinct contracts.

**Suicidal** In Figure 9, the overlap in identifying the Suicide vulnerability is depicted, a flaw that allows an attacker to terminate a smart contract. Mythril, Maian, and Vandal are capable of detecting this vulnerability, whereas Oyente is incapable of recognizing it in bytecode. Vandal, consistent with its performance in detecting other vulnerabilities, flags the most significant number of contracts as susceptible. Notably, the shared detection by all three tools is more substantial in this case compared to the Reentrancy vulnerability. Additionally, there is a significant overlap between Maian and Vandal, with both identifying the issue in approximately 14,991 contracts. Despite this, there remains a considerable disparity in the assessments across most contracts by the three analyzed tools.

**Unchecked Call** Unchecked calls can be exploited by an attacker to induce undefined behavior in a smart contract. This vulnerability is identifiable by Mythril and Vandal. Figure 10 displays the intersection of these tools in detecting the issue within bytecode. We see Vandal's tendency to report excessively, in this case identifying 1,771,577 instances as vulnerable, which is over one-fourth of all contracts. Further, there is a significant degree of overlap with Mythril's find-

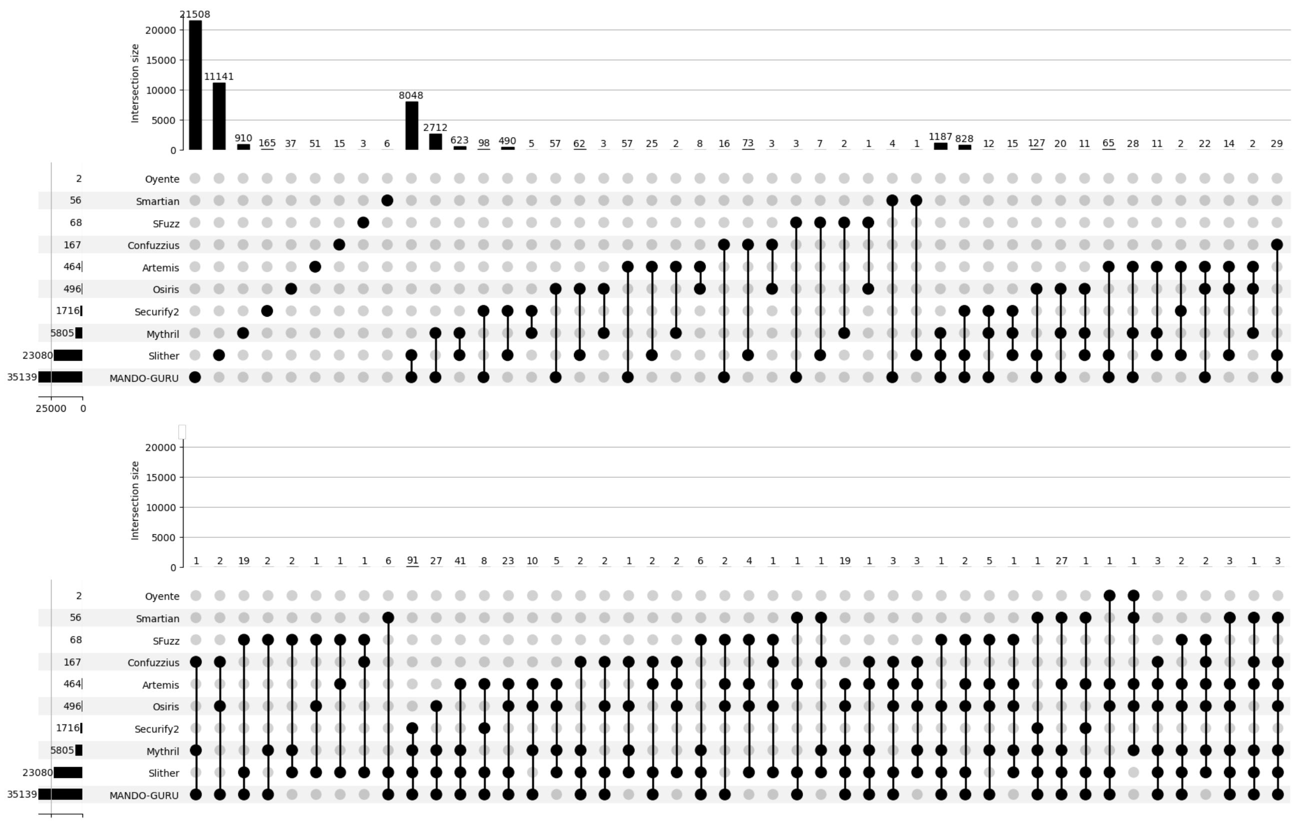


Figure 2: Overlap of source code-based tools detecting the *Reentrancy* vulnerability.

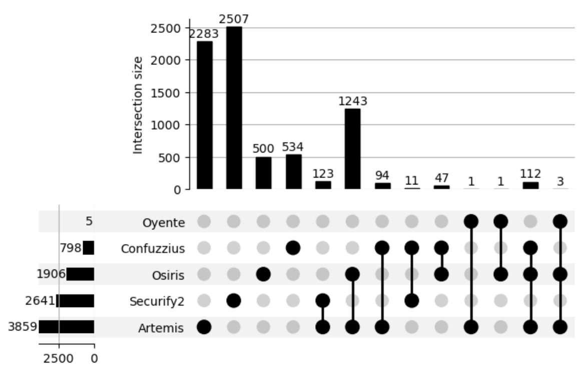


Figure 3: Overlap of source code-based tools detecting the *ToD* vulnerability.

ings. Nevertheless, this overlap constitutes less than $0.5\,\%$ of the total detections of Vandal.

**TxOrigin** The use of txOrigin gives an attacker the potential to bypass authorization checks in a smart contract. The overlap between Mythril and Vandal in detecting this issue is illustrated in Figure 11. The two tools agree on the vulnerability in only 2,372 contracts, representing less than $5\,\%$ of the total contracts in our dataset. Collectively, these tools have flagged $1.3\,\%$ of contracts, indicating a possible overestimation of this vulnerability when all tools are utilized together.

**Time Dependency** The intersection of Oyente and Mythril in detecting the Time Dependency vulnerability is depicted in Figure 12. These tools agree on the vulnerability in 8,735 contracts. However, they diverge in their assessments for over $90\,\%$ of the contracts they flagged, indicating a significant discrepancy in their detection capabilities for this particular vulnerability.

**Summary** Overall, there is a parallel pattern observed with bytecode analysis tools compared to those analyzing source code. Consistently, there's a notable disagreement among the tools regarding the majority of contracts and types of vulnerabilities. This pattern suggests that these tools may not be highly effective on a larger scale. Although this quantitative analysis doesn't fully capture the exact performance of the bytecode tools, the evident discrepancies among various tools are clear.

## 4.4. Source Code vs. Bytecode

We consider two vulnerability scanners, Mythril and Maian, that allow the input of source code and bytecode. In this section, we evaluate the performance difference between these scanners and see whether they have a performance difference being exposed to source code or bytecode. This analysis is based on our quantitative datasets.

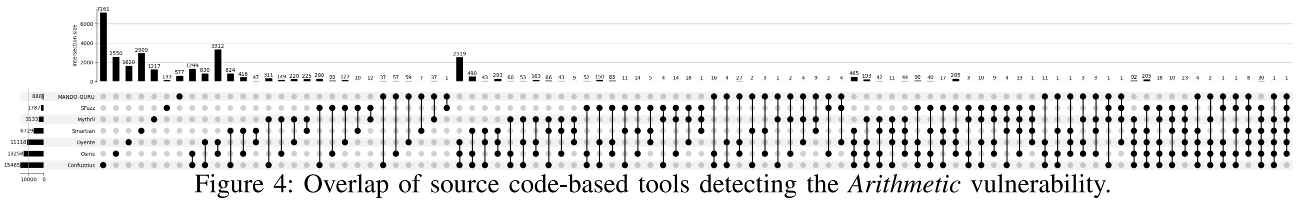

Figure 4: Overlap of source code-based tools detecting the *Arithmetic* vulnerability.

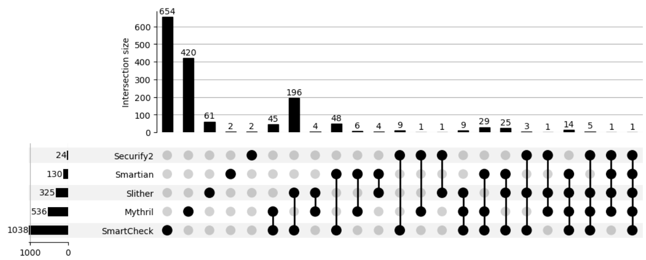

Figure 5: Overlap of source code-based tools detecting the *txOrigin* vulnerability.

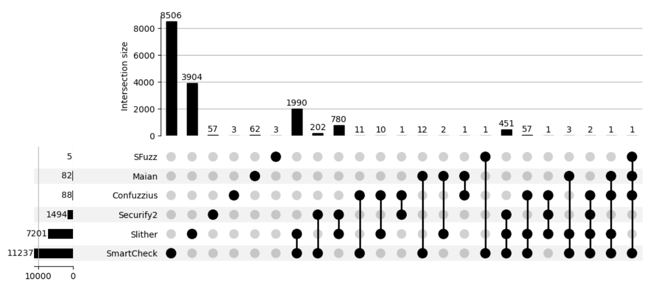

Figure 6: Overlap of tools detecting the *Locked Ether* vulnerability.

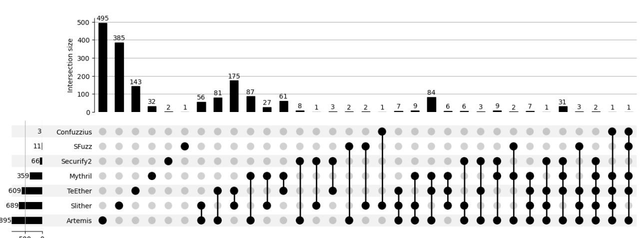

Figure 7: Overlap of tools detecting the *DelegateCall* vulnerability.

**Maian** flags four smart contracts as Suicidal in source code and 93 in bytecode, overlapping in three cases. It also finds 80 contracts with locked ether in source code and 3,223 in bytecode, with 66 common detections. Generally, Maian shows a higher positive rate in bytecode analysis.

**Mythril** shows no common detections between source code and bytecode for the DelegateCall and Reentrancy vulnerabilities and only one shared detection for the Suicide vulnerability. It finds an overlap of 35 in Arithmetic Bugs, 76 in txOrigin, and a notable 1,242 in Time Dependency. Contrary to Maian, Mythril generally presents a higher positive detection rate in source codes compared to bytecodes.

### 4.5. Qualitative Analysis based on RGT Dataset

In this section, we evaluate both source code and bytecode analysis tools using our meticulously curated reentrancy dataset RGT, as detailed in Section 3.3. As mentioned, this dataset is exclusively annotated for the reentrancy vulnerability, which is categorized into three subtypes: call, send, and transfer.

Our analysis covers two distinct approaches: the first includes only the 'call' subtype under the umbrella of reentrancy, while the second encompasses all three subtypes. This dual approach aids in determining the specific type of reentrancy each tool is geared to identify, especially since the tools do not explicitly state their focus on certain subtypes.

**Analysis of Source Code-based Tools** We present our findings for the call subtype in Table 3, which displays evaluation results for nine vulnerability scanners. In contrast to the quantitative analysis, we do not consider samples if the applied vulnerability scanner does not finish the evaluation.

Securify2 emerged as the most effective tool with an F1-Score of 38%, but it was only able to test 662 instances. The key observation here is the generally poor detection rate for the call subtype across all tools.

| Tools | TN | FP | FN | TP | F1 | Acc |
|---|---|---|---|---|---|---|
| Artemis | 5,174 | 95 | 358 | 71 | 0.24 | 0.92 |
| Osiris | 1,727 | 81 | 112 | 49 | 0.34 | 0.90 |
| Confuzzius | 2,956 | 21 | 159 | 30 | 0.25 | 0.94 |
| Smartian | 2,279 | 3 | 168 | 10 | 0.10 | 0.93 |
| Mythril | 4,069 | 398 | 298 | 83 | 0.19 | 0.86 |
| sFuzz | 617 | 24 | 34 | 7 | 0.19 | 0.91 |
| Oyente | 2,854 | 0 | 169 | 1 | 0.01 | 0.94 |
| Slither | 4,469 | 1,483 | 41 | 411 | 0.35 | 0.76 |
| Securify2 | 603 | 41 | 4 | 14 | 0.38 | 0.93 |
| MANDO-GURU | 332 | 4,027 | 4 | 281 | 0.12 | 0.13 |

TABLE 3: Results of Ground Truth call-subtype Reentrancy dataset with source code tools.

The results for all reentrancy subtypes are presented in Table 4. Here, we observe a general decline in detection effectiveness among most tools. However, four tools – Slither, Securify2, Mythril, and MANDO-GURU – show improved performance, suggesting their emphasis on all three reentrancy bug subtypes.

This analysis reveals varied interpretations of 'reentrancy' across tools. Most prioritize the call subtype, but some also regard send and transfer subtypes as important.

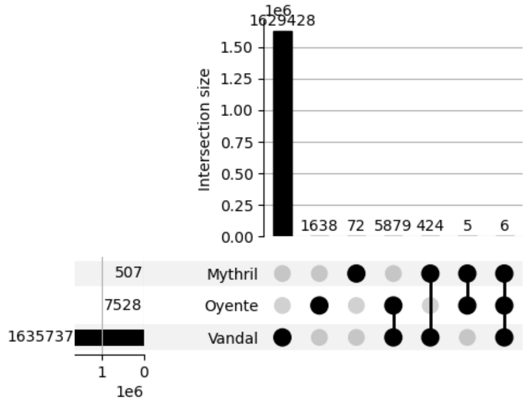


Figure 8: Overlap of bytecode tools detecting the reentrancy vulnerability.

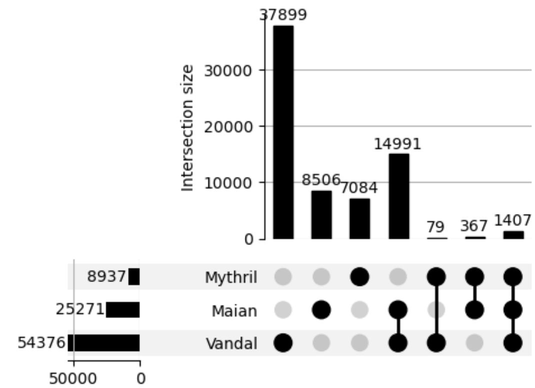


Figure 9: Overlap of bytecode tools detecting the suicidal vulnerability.

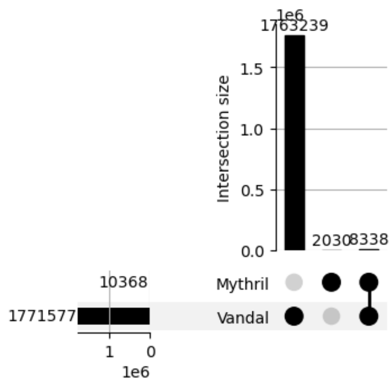


Figure 10: Overlap of bytecode tools detecting the unchecked call vulnerability.

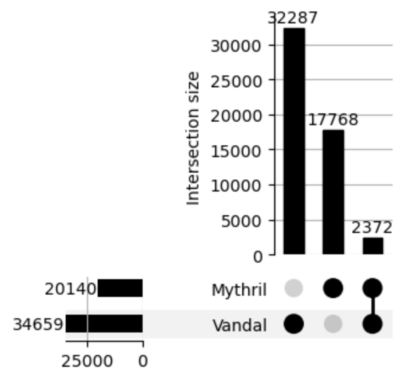


Figure 11: Overlap of bytecode tools detecting the txOrigin vulnerability.

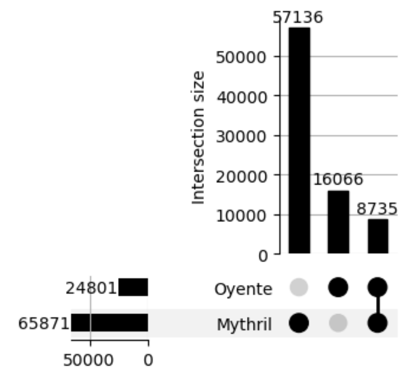


Figure 12: Overlap of bytecode tools detecting the Time Dependency vulnerability.

| Tools | TN | FP | FN | TP | F1 | Acc |
|---|---|---|---|---|---|---|
| Artemis | 4,378 | 53 | 1,154 | 113 | 0.16 | 0.79 |
| Osiris | 1,257 | 46 | 582 | 84 | 0.21 | 0.68 |
| Confuzzius | 2,484 | 14 | 631 | 37 | 0.10 | 0.80 |
| Smartian | 1,931 | 2 | 516 | 11 | 0.04 | 0.79 |
| Mythril | 3,517 | 298 | 850 | 183 | 0.24 | 0.76 |
| sFuzz | 411 | 17 | 240 | 14 | 0.10 | 0.62 |
| Oyente | 2,279 | 0 | 744 | 1 | 0.00 | 0.75 |
| Slither | 4,324 | 704 | 186 | 1,190 | 0.73 | 0.86 |
| Securify2 | 592 | 24 | 15 | 31 | 0.61 | 0.94 |
| MANDO-GURU | 321 | 3,510 | 15 | 798 | 0.31 | 0.24 |

TABLE 4: Results of Ground Truth all subtypes Reentrancy dataset with source code tools.

This variance in definitions hinders direct performance comparisons between tools.

**Analysis of Bytecode-based Tools** As we already explained in Section 3.3, we do not compile smart contracts of our dataset. Instead, we use their addresses to download the bytecode from the blockchain. This allows us to analyze the deployed smart contract version without introducing discrepancies due to compiler versions and optimizations.

The outcomes of bytecode tools for the call-subtype reentrancy are presented in Table 5. It's observed once more that bytecode tools are capable of analyzing a greater number of contracts compared to their source code counterparts. Additionally, it's notable that Mythril's performance is less effective on bytecode than on source code, which is understandable considering the richer information available in the source code. Vandal, despite achieving the highest F1-score among the tools, falls behind in terms of accuracy compared to the others. Interestingly, Oyente demonstrates improved performance on bytecode over its analysis of the same data in source code. Overall, the effectiveness of these tools in analyzing bytecode for call-subtype reentrancy is not particularly impressive.

| Tools | TN | FP | FN | TP | F1 | Acc |
|---|---|---|---|---|---|---|
| Vandal | 3,662 | 2,292 | 107 | 345 | 0.22 | 0.63 |
| Oyente | 5,900 | 54 | 412 | 40 | 0.15 | 0.93 |
| Mythril | 5,953 | 1 | 452 | 0 | 0.00 | 0.93 |

TABLE 5: Results of Ground Truth call-subtype Reentrancy dataset with bytecode tools.

The findings for all reentrancy subtypes are presented in Table 6. In this comparison, both Oyente and Mythril exhibit diminished performance. On the other hand, Vandal improves its F1-score significantly, jumping from 22 % to 40 %. However, it's important to note that Vandal's accuracy decreases by one percent.

| Tools | TN | FP | FN | TP | F1 | Acc |
|---|---|---|---|---|---|---|
| Vandal | 3,194 | 1,836 | 575 | 801 | 0.40 | 0.62 |
| Oyente | 4,995 | 35 | 1,317 | 59 | 0.08 | 0.79 |
| Mythril | 5,029 | 1 | 1,376 | 0 | 0.00 | 0.79 |

TABLE 6: Results of Ground Truth all subtypes Reentrancy dataset with bytecode tools.

**Summary.** Ultimately, Slither had the best F1-Score overall on our handcrafted reentrancy dataset with 73 % for the source code-based analysis. Although bytecode analysis tools demonstrated a higher completion rate, their detection efficacy was notably lower compared to when the vulnerability scanners were applied directly to the source code. A key insight from this study is the significant influence that vulnerability scanners' definitions of a vulnerability have on the performance of various tools on the same dataset.

Our investigation focused on the nuances of the reentrancy bug, a well-recognized type of vulnerability. However, it's plausible that similar discrepancies in the definition of vulnerabilities in different studies could also affect the detection of other types of vulnerabilities.

## 4.6. Qualitative Analysis based on AGT Dataset

In our Audit dataset, we ensure that all smart contracts are compiled using the required compiler version, facilitating the analysis of their bytecode. On the other hand, when it comes to source code-based analysis, we encounter limitations. Due to the small number of vulnerable contracts and the inability of tools to analyze them without modifications to either the project's source code or the vulnerability scanners, we couldn't conduct source code-based analysis effectively. Consequently, our focus in this section is primarily on the analysis of bytecode.

**Reentrancy** We show the results of the reentrancy bug detection in Table 7. Like in the case of our self-labeled dataset RGT (cf. Section 3.3), all three analyzed tools show low detection performance.

| Tools | TN | FP | FN | TP | F1 | Acc |
|---|---|---|---|---|---|---|
| Vandal | 87 | 31 | 31 | 36 | 0.40 | 0.53 |
| Oyente | 165 | 0 | 67 | 0 | 0.00 | 0.71 |
| Mythril | 163 | 2 | 67 | 0 | 0.00 | 0.70 |

TABLE 7: Results of Audit dataset for Reentrancy with bytecode tools.

**Other Vulnerabilities.** Our additional findings are detailed in Table 8. In this table, we present the tools used and the corresponding SWC identifiers for each vulnerability, as referenced in Section 2.2. These results further highlight the limited effectiveness of all involved tools in detecting various vulnerabilities.

| Tools | TN | FP | FN | TP | F1 | Acc | Vulnerability (SWC) |
|---|---|---|---|---|---|---|---|
| Mythril | 187 | 18 | 25 | 2 | 0.09 | 0.81 | Int Over/Underflow (101) |
| Vandal | 52 | 28 | 90 | 62 | 0.51 | 0.49 | Unchecked Call (104) |
| Oyente | 216 | 0 | 16 | 0 | 0.00 | 0.93 | ToD (114) |
| Mythril | 202 | 18 | 9 | 3 | 0.18 | 0.88 | Time Dependency (116) |
| Oyente | 220 | 0 | 12 | 0 | 0.00 | 0.95 | Time Dependency (116) |

TABLE 8: Results of Audit dataset for different vulnerabilities with bytecode tools.

## 4.7. Scanning Robustness

In this section, we assess the robustness of the vulnerability scanners. Given that some scanners are unable to complete the analysis of certain smart contracts, our focus is on the frequency of successful analyses. This is in the interest of smart contract developers since an unfinished scan leaves them with some degree of uncertainty.

Figure 13 shows the percentage of how many contracts were successfully scanned by the different source code-based tools. We can see that only five tools (Artemis, Mythril, Slither, GNN, MANDO-GURU, and Smartcheck) were able to at least check half of the smart contracts in our dataset. Otherwise, the tools quite often were not able to complete their analysis.

We attribute these disheartening results to several factors. First, most tools base their analysis on specific compiler versions of the Solidity compiler. Since the area of smart contracts is a rather fast-growing field, compiler versions are regularly updated and at some points introduce breaking changes. A further concern involves reliance on third-party tools, like constraint-solving algorithms. These tools frequently receive updates, which might lead to compatibility issues with existing versions of the tools or Solidity. Further, some smart contracts are written in another language. However, most tools are only able to analyze Solidity code. Generally, these effects can be attributed to the effect of software aging. Most pressing is the issue of dependency on specific compiler versions, which makes most of the tooling dependent on a fast-changing piece of code.

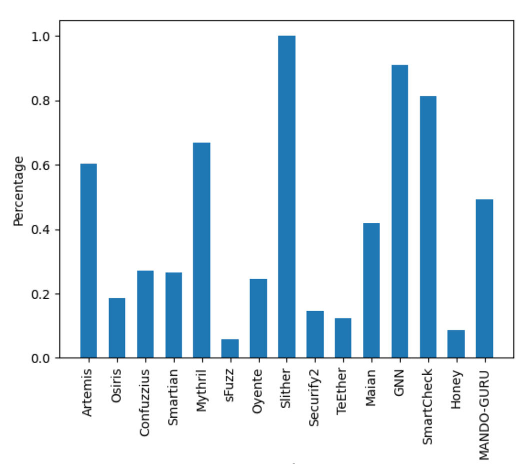

Figure 13: Robustness of Tool Detection

## 5. Discussion

In this section, we engage in a focused discussion about our study's outcomes. We first analyze why the tools used show discrepancies in their results, considering their different methodologies and biases. Next, we debate if using more tools leads to improved vulnerability detection, balancing the pros and cons of diverse methodologies against potential complexities. We then evaluate the ease of setup and use for these tools, considering their user-friendliness and technical requirements. Next, we assess the commonness of vulnerabilities in smart contracts, understanding their prevalence and impact on the security and efficacy of these contracts. Lastly, we discuss the limitations of our study in terms of both the methodology employed and the datasets utilized. This discussion aims to provide a comprehensive understanding of our findings, their implications, and limitations in the practical use of smart contracts vulnerability scanners.

### 5.1. Disagreement of Tools

The disagreement among tools is largely attributed to the varying times at which they were developed, leading to three key issues which we discuss below.

**Compiler Version Changes** Tools, particularly those analyzing source code, are often designed for specific versions of the Solidity compiler. As the compiler evolves, these tools may not be updated promptly to accommodate new versions or resolve potential breaking changes.

**Evolution of Smart Contract Programming** The programming practices in smart contracts can shift over time. For instance, the use of 'send'/'transfer' calls was initially common in smart contracts. However, changes in Ethereum, like the gas limit alterations for these calls, have led to programmers being advised against their use.

**New Developments and Attack Variations** The emergence of new developments and variations in attack strategies can alter the landscape of vulnerabilities. An example is the introduction of built-in defenses against integer overflows and underflows in the compiler.

Additionally, vulnerability definition can differ across tools, influenced by the authors' perspectives and interpretations of what constitutes a vulnerability. In some cases, vulnerabilities might be part of a multi-step exploit process.

Overall, the significant disagreement among tools is concerning. It suggests that their detection capabilities are not yet robust enough to eliminate bugs effectively, a conclusion further reinforced by the ongoing exploits occurring on the blockchain.

### 5.2. Usage of Multiple Tools

Using a combination of tools to identify vulnerabilities in a smart contract might seem like a promising strategy. The differing results from various tools suggest that their combined use could potentially offer a more thorough analysis. However, this method has its drawbacks, primarily the increase in positive detections, which includes false positives. This means that more extensive portions of the smart contract would require examination.

Moreover, the challenge lies in selecting an appropriate mix of tools. There's no definitive or universally effective combination that guarantees the successful detection of all vulnerabilities. Each tool has its own strengths, weaknesses, and focus areas, and their efficiency can vary depending on the contract's specific features and vulnerabilities.

In summary, there is no foolproof or 'silver bullet' method for detecting all vulnerabilities in smart contracts. While using multiple tools might enhance the breadth of analysis, it does not necessarily lead to flawless detection and can complicate the assessment process.

### 5.3. Difficulty of Usage

The implementation ease and use of smart contract vulnerability detection tools vary widely. Some tools are easily operated with a Docker command, while others require complex dependency resolutions, especially Java-based tools using Maven. Running these tools often necessitates specific configurations and parameters, adding complexity and reducing practicality for developers.

Further, we discuss a range of smart contract analysis tools, which were not included in our analysis due to their source code being inaccessible. These tools encompass EthPloit [78], Harvey [79], ReGuard [80], Solar [81], SmartScopy [82], SESCon [83] and sCompile [84], Ether* S-gram [85], Sereum [86], Easyflow [87], Zeus [88], Ethainter [89] and the framework described in [90]. Similar considerations apply to SACS [91] and Sailfish [92]. Despite assurances of making their code open source, these tools remain unavailable as of the time this paper was written.

On a different note, while the machine learning model underlying ESCORT [23] is not accessible, the tools from which ESCORT derives its learning are available and are consequently incorporated into our analysis.

In our analysis, we intended to include a wider range of tools but faced challenges due to issues like complex setups and outdated functionalities. Tools such as ContractFuzzer [93], EtherRacer [94], SODA [95], and SecurifyV1 [19] were non-functional because of outdated or unmaintained repositories, though we could include the updated SecurifyV2.

NeuCheck [96] lacked clear setup instructions, making it unsuitable for our study. Tools like eThor [97] and EthBMC [62], requiring intensive constraint solving, had prohibitive execution times for large-scale analysis.

Lastly, we excluded tools such as Manticore [98] and MadMax [99], which focus on metrics like code coverage and gas issues, respectively, not aligning with our vulnerability detection objective.

### 5.4. Limitations

Our study is subject to certain limitations concerning the datasets and methodology.

The datasets employed were collected before the application of various scanning tools, meaning our datasets lack the most recent contracts that might exhibit newer coding styles. Nonetheless, the scanners applied in our study were developed with the capability to analyze smart contracts, which should suffice for analyzing our datasets.

Moreover, our self-labeled RGT dataset inherently carries a bias in defining reentrancy vulnerabilities. To counteract this, we included three different subtypes of the reentrancy vulnerability in our analysis (cf. Section 4.5). A similar challenge is faced with the AGT dataset, which was labeled based on audit reports, introducing another potential labeling bias by the audit authors.

Given the extensive scope of our analysis, we limited the analysis time for each tool, which may restrict the effectiveness of certain tools, especially fuzzing tools, that might perform better with extended runtimes. This compromise was deemed necessary to facilitate a large-scale analysis.

Acknowledging the possibility of inaccuracies in our analysis, labeling, and tooling, we intend to make our datasets and scripts publicly available at https://github.com/sss-wue/sc-study/. This will allow future researchers to explore potential discrepancies and validate our results.

For future research, an interesting aspect would be to explore the specific vulnerability definitions utilized by various tools. Our examination of the RGT dataset revealed minor variations in how different tools define vulnerabilities.

## 6. Related Work

The related work concerning analysis of smart contract vulnerability detection tools can be categorized into two categories: Theoretical and practical analysis.

**Theoretical Analysis** Several studies including [29]–[34] have conducted theoretical analyses of smart contract detection tools and vulnerabilities in Ethereum smart contracts. These studies primarily focus on categorizing vulnerabilities, listing available tools, and discussing their properties without performing actual evaluations or comparisons. In summary, these studies provide theoretical insights into smart contract vulnerabilities and detection tools, but they do not offer practical assessments of how well these tools perform in real-world scenarios.

**Practical Analysis** The SmartBugs framework [36] compares ten smart contract analysis tools using a dataset of 143 annotated contracts.

The Solidify framework [38] assesses six static analysis tools on 50 contracts injected with 9,369 bugs, with each vulnerability randomly represented in the code.

Ren et al. [39] analyzed nine tools out of the three categories: static analysis, symbolic execution, and dynamic fuzzing. Their study utilized a dataset that encompassed real-world contracts, manually injected bugs, and verified vulnerable contracts, culminating in a total of 46,186 unique contracts, of which 214 were confirmed as vulnerable. In contrast to our research, their investigation was solely concentrated on reentrancy vulnerabilities in source codes.

Kushwaha et al. [40] performed a theoretical comparison of 86 analysis tools, as documented in 145 research papers. From this survey, they selected 16 tools for analysis, primarily focusing on categories such as symbolic execution and constraint solving. Their analysis was based on a relatively limited dataset comprising only 30 contracts, tagged with five specific vulnerabilities.

Dika and Nowostawski [41] provided insights into four tools using a dataset of 45 contracts, split between 21 clean and 24 vulnerable.

He et al. [100] focused on random number vulnerabilities in Fomo3d-like games and discussed three auditing tools for smart contract security.

Peng et al. [42] presented an overview of 29 smart contract analysis tools, assessing their language support, analysis methods, and detectable vulnerabilities. They compared five tools using a dataset of 300 randomly collected smart contracts from Etherscan.

While previous studies, provide practical evaluations, they are constrained in scope, primarily focusing on a narrow selection of tools or utilizing relatively small datasets for comparison. Notably, none of these surveys address the intersection of tools, a significant oversight in the related literature. This gap is crucial as it reveals considerable discrepancies in results across almost every tool, highlighting the importance of comprehensive and varied datasets for robust tool evaluation.

## 7. Conclusion

To conclude, our extensive analysis, encompassing millions of smart contracts with both source codes and bytecodes, including those that are manually labeled, highlights a clear finding: there is substantial scope for enhancement in the realm of smart contract security. This study underscores the ongoing and complex nature of the challenge of detecting vulnerabilities effectively.

## References

[1] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” Accessed 2023. [Online]. Available: http://bitcoin.org/bitcoin.pdf

[2] “Ethereum Whitepaper,” Accessed 2023. [Online]. Available: https://ethereum.org/en/whitepaper/

[3] “Hyperledger project,” Accessed 2023. [Online]. Available: https://www.hyperledger.org/

[4] “What’s a DApp?” Accessed 2023. [Online]. Available: https://www.bitcoin.com/get-started/what-is-a-dapp/

[5] “What is decentralized finance (DeFi)?” Accessed 2023. [Online]. Available: https://www.techtarget.com/whatis/definition/decentralized-finance-DeFi

[6] “Non-Fungible Token (NFT): What It Means and How It Works,” Accessed 2023. [Online]. Available: https://www.investopedia.com/non-fungible-tokens-nft-5115211

[7] “CryptoKitties,” Accessed 2023. [Online]. Available: https://www.cryptokitties.co/

[8] “Tornado.cash,” Accessed 2023. [Online]. Available: https://tornadocash.eth.link/

[9] "MakerDAO," Accessed 2023. [Online]. Available: https://makerdao.com/en/

[10] "Uniswap," Accessed 2023. [Online]. Available: https://uniswap.org/

[11] "MakerDAO expected to generate $105 million in profits in 2024, Maker price nearly rallies by 8%," Accessed 2024. [Online]. Available: https://www.fxstreet.com/cryptocurrencies/news/makerdao-expected-to-generate-105-million-in-profits-in-2024-maker-price-nearly-rallies-by-8-202312290205

[12] D. Siegel, "Understanding the DAO Attack," Accessed 2023. [Online]. Available: https://www.coindesk.com/understanding-dao-hack-journalists

[13] "Explained: The SafeMoon Hack," Accessed 2023. [Online]. Available: https://www.halborn.com/blog/post/explained-the-safe-moon-hack-march-2023

[14] "Explained: The LendHub Hack," Accessed 2023. [Online]. Available: https://www.halborn.com/blog/post/explained-the-lendhub-hack-january-2023

[15] "How Deus Finance Was Exploited for $13.4M on Fantom," Accessed 2023. [Online]. Available: https://www.coindesk.com/tech/2022/04/28/how-deus-finance-was-exploited-for-134m-on-fantom/

[16] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," 2018.

[17] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[18] J. Krupp and C. Rossow, "teEther: Gnawing at ethereum to automatically exploit smart contracts," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1317–1333. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[19] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–82. [Online]. Available: https://doi.org/10.1145/3243734.3243780

[20] J. Feist, G. Grieco, and A. Groce, "Slither: A static analysis framework for smart contracts," *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pp. 8–15, 2019.

[21] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *2018 IEEE/ACM 1st International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2018, pp. 9–16.

[22] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." in *IJCAI*, 2020, pp. 3283–3290.

[23] C. Sendner, H. Chen, H. Fereidooni, L. Petzi, J. König, J. Stang, A. Dmitrienko, A.-R. Sadeghi, and F. Koushanfar, "Smarter contracts: Detecting vulnerabilities in smart contracts with deep transfer learning," in *Proceedings of the 2023 Network and Distributed System Security (NDSS) Symposium*, ser. NDSS '23, 01 2023.

[24] O. Lutz, H. Chen, H. Fereidooni, C. Sendner, A. Dmitrienko, A. R. Sadeghi, and F. Koushanfar, "Escort: ethereum smart contracts vulnerability detection using deep neural network and transfer learning," *arXiv preprint arXiv:2103.12607*, 2021.

[25] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021, pp. 103–119.

[26] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "Sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 778–788. [Online]. Available: https://doi.org/10.1145/3377811.3380334

[27] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic dataflow analyses," in *Proceedings of the International Conference on Automated Software Engineering*, 2021.

[28] "Crypto Hacks 2023: Full List Of Scams And Exploits As Millions Go Missing," Accessed 2023. [Online]. Available: https://www.ccn.com/education/crypto-hacks-2023-full-list-of-scams-and-exploits-as-millions-go-missing/

[29] X. Tang, K. Zhou, J. Cheng, H. Li, and Y. Yuan, "The vulnerabilities in smart contracts: A survey," in *Advances in Artificial Intelligence and Security: 7th International Conference, ICAIS 2021, Dublin, Ireland, July 19-23, 2021, Proceedings, Part III 7*. Springer, 2021, pp. 177–190.

[30] H. Rameder, M. Di Angelo, and G. Salzer, "Review of automated vulnerability analysis of smart contracts on ethereum," *Frontiers in Blockchain*, vol. 5, p. 814977, 2022.

[31] H. Zhou, A. Milani Fard, and A. Makanju, "The state of ethereum smart contracts security: Vulnerabilities, countermeasures, and tool support," *Journal of Cybersecurity and Privacy*, vol. 2, no. 2, pp. 358–378, 2022.

[32] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities: a survey," *arXiv preprint arXiv:1908.08605*, 2019.

[33] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart contract: Attacks and protections," *IEEE Access*, vol. 8, pp. 24 416–24 427, 2020.

[34] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Systematic review of security vulnerabilities in ethereum blockchain smart contract," *IEEE Access*, vol. 10, pp. 6605–6621, 2022.

[35] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International conference on software engineering*, 2020, pp. 530–541.

[36] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," in *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, 2020, pp. 1349–1352.

[37] M. di Angelo, T. Durieux, J. F. Ferreira, and G. Salzer, "Smartbugs 2.0: An execution framework for weakness detection in ethereum smart contracts," *arXiv preprint arXiv:2306.05057*, 2023.

[38] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 415–427.

[39] M. Ren, Z. Yin, F. Ma, Z. Xu, Y. Jiang, C. Sun, H. Li, and Y. Cai, "Empirical evaluation of smart contract testing: What is the best choice?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 566–579.

[40] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Ethereum smart contract analysis tools: A systematic review," *IEEE Access*, vol. 10, pp. 57 037–57 062, 2022.

[41] A. Dika and M. Nowostawski, "Security vulnerabilities in ethereum smart contracts," in *2018 IEEE international conference on Internet of Things (iThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2018, pp. 955–962.

[42] P. Qian, Z. Liu, Q. He, B. Huang, D. Tian, and X. Wang, "Smart contract vulnerability detection technique: A survey," *arXiv preprint arXiv:2209.05872*, 2022.

[43] "Ethereum (eth) blockchain explorer," Accessed 2023. [Online]. Available: https://etherscan.io/

[44] "InterPlanetary File System (IPFS)," Accessed 2023. [Online]. Available: https://ipfs.tech/

[45] "Testnet Goerli," Accessed 2023. [Online]. Available: https://goerli.net/

[46] "Testnet Rinkeby," Accessed 2023. [Online]. Available: https://www.rinkeby.io/

[47] "Testnet Ropsten," Accessed 2023. [Online]. Available: https://ropsten.etherscan.io/

[48] "Testnet Kovan," Accessed 2023. [Online]. Available: https://kovan-testnet.github.io/website/

[49] A. Wang, H. Wang, B. Jiang, and W. K. Chan, "Artemis: An improved smart contract verification tool for vulnerability detection," in *2020 7th International Conference on Dependable Systems and Their Applications (DSA)*, 2020, pp. 173–181.

[50] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 664–676. [Online]. Available: https://doi.org/10.1145/3274694.3274737

[51] B. Mueller, "Smashing ethereum smart contracts for fun and real profit," in *9th Annual HITB Security Conference (HITBSecConf)*, Amsterdam, Netherlands, 2018.

[52] H. H. Nguyen, N.-M. Nguyen, H.-P. Doan, Z. Ahmadi, T.-N. Doan, and L. Jiang, "Mando-guru: Vulnerability detection for smart contract source code by heterogeneous graph embeddings," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 11 2022, pp. 1736–1740. [Online]. Available: https://doi.org/10.1145/3540250.3558927

[53] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," 2018.

[54] "The Solidity Contract-Oriented Programming Language," Accessed 2023. [Online]. Available: https://github.com/ethereum/solidity

[55] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, 2014.

[56] SmartContractSecurity, "Smart contract weakness classification and test cases," Accessed 2023. [Online]. Available: https://swcregistry.io/

[57] ——, "Swc-110 assert violation," Accessed 2023. [Online]. Available: https://swcregistry.io/docs/SWC-110/

[58] H. Ermawan, "Vulnerabilities and attacks of smart contracts." Accessed 2023. [Online]. Available: https://medium.com/hryer-dev/vulnerabilities-attacks-of-smart-contracts-9f112ea6c52c

[59] SmartContractSecurity, "Swc-114 example of tod vulnerability," Accessed 2023. [Online]. Available: https://swcregistry.io/docs/SWC-114

[60] I. Grishchenko, M. Maffei, and C. Schneidewind, "Ethertrust: Sound static analysis of ethereum bytecode," 2018.

[61] Ethereum Foundation and chainSecurity, "Securify v2.0," https://github.com/eth-sri/securify2, 2023, gitHub repository.

[62] J. Frank, C. Aschermann, and T. Holz, "ETHBMC: A bounded model checker for smart contracts," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2757–2774. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/frank

[63] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 531–548. [Online]. Available: https://doi.org/10.1145/3319535.3363230

[64] M. Zalewski, "American fuzzy lop," https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016, whitepaper.

[65] Y. Zhuang, Z. Liu, P. Qian, Q. Liu, X. Wang, and Q. He, "Smart contract vulnerability detection using graph neural network." in *IJCAI*, 2020, pp. 3283–3290.

[66] "Google BigQuery," Accessed 2023. [Online]. Available: https://cloud.google.com/bigquery

[67] "Erigon," Accessed 2023. [Online]. Available: https://github.com/ledgerwatch/erigon

[68] "Geth," Accessed 2023. [Online]. Available: https://geth.ethereum.org/

[69] "Introduction — Web3.py 5.12.1 documentation," Accessed 2023. [Online]. Available: https://web3py.readthedocs.io/en/stable/

[70] "Smartbugs wild dataset," Accessed 2023. [Online]. Available: https://github.com/smartbugs/smartbugs-wild

[71] "Quantstamp Security Audits," Accessed 2023. [Online]. Available: https://certificate.quantstamp.com/

[72] "OpenZeppelin Security Audits," Accessed 2023. [Online]. Available: https://blog.openzeppelin.com/security-audits/

[73] "Trail of Bits Security Audits," Accessed 2023. [Online]. Available: https://github.com/trailofbits/publications/tree/master/reviews

[74] "ConsenSys Audits," Accessed 2023. [Online]. Available: https://consensys.net/diligence/audits/

[75] "CertiK Audits," Accessed 2023. [Online]. Available: https://www.certik.com/

[76] "BeeGFC," Accessed 2023. [Online]. Available: https://www.beegfs.io

[77] "Docker," Accessed 2023. [Online]. Available: https://www.docker.com/

[78] Q. Zhang, Y. Wang, J. Li, and S. Ma, "Ethploit: From fuzzing to efficient exploit generation against smart contracts," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 116–126.

[79] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1398–1409.

[80] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 65–68.

[81] A. Li and F. Long, "Detecting standard violation errors in smart contracts," *arXiv preprint arXiv:1812.07702*, 2018.

[82] Y. Feng, E. Torlak, and R. Bodik, "Precise attack synthesis for smart contracts," *arXiv preprint arXiv:1902.06067*, 2019.

[83] A. Ali, Z. U. Abideen, and K. Ullah, "Sescon: Secure ethereum smart contracts by vulnerable patterns' detection," *Security and Communication Networks*, vol. 2021, pp. 1–14, 2021.

[84] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "scompile: Critical path identification and analysis for smart contracts," in *Formal Methods and Software Engineering: 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5–9, 2019, Proceedings 21*. Springer, 2019, pp. 286–304.

[85] H. Liu, C. Liu, W. Zhao, Y. Jiang, and J. Sun, "S-gram: towards semantic-aware security auditing for ethereum smart contracts," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, 2018, pp. 814–819.

[86] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

[87] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "Easyflow: Keep ethereum away from overflow," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings*, ser. ICSE '19. IEEE Press, 2019, p. 23–26. [Online]. Available: https://doi.org/10.1109/ICSE-Companion.2019.00029

[88] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: analyzing safety of smart contracts," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[89] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: A smart contract security analyzer for composite vulnerabilities," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 454–469. [Online]. Available: https://doi.org/10.1145/3385412.3385990

[90] N. F. Samreen and M. H. Alalfi, "Reentrancy vulnerability identification in ethereum smart contracts," in *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. IEEE, feb 2020. [Online]. Available: https://doi.org/10.1109%2Fiwbose50093.2020.9050260

[91] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*. IEEE, 2018, pp. 1–5.

[92] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 161–178.

[93] B. Jiang, Y. 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*, 2018, pp. 259–269.

[94] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," in *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, 2019, pp. 363–373.

[95] T. Chen, R. Cao, T. Li, X. Luo, G. Gu, Y. Zhang, Z. Liao, H. Zhu, G. Chen, Z. He *et al.*, "Soda: A generic online detection framework for smart contracts." in *NDSS*, 2020.

[96] N. Lu, B. Wang, Y. Zhang, W. Shi, and C. Esposito, "Neucheck: A more practical ethereum smart contract security analysis tool," *Software: Practice and Experience*, vol. 51, no. 10, pp. 2065–2084, 2021.

[97] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "Ethor: Practical and provably sound static analysis of ethereum smart contracts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 621–640. [Online]. Available: https://doi.org/10.1145/3372297.3417250

[98] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[99] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Analyzing the out-of-gas world of smart contracts," *Commun. ACM*, vol. 63, no. 10, p. 87–95, sep 2020. [Online]. Available: https://doi.org/10.1145/3416262

[100] D. He, Z. Deng, Y. Zhang, S. Chan, Y. Cheng, and N. Guizani, "Smart contract vulnerability analysis and security audit," *IEEE Network*, vol. 34, no. 5, pp. 276–282, 2020.

# Appendix A. Additional Figures

Figure 14 shows our analysis of the Time Dependency vulnerability on our source code dataset SCD.

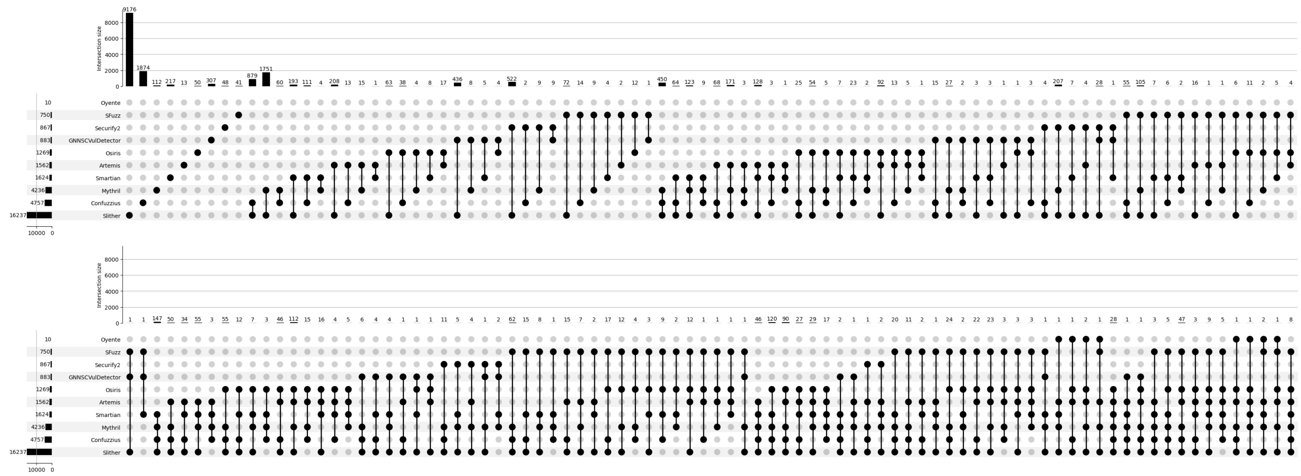


Figure 14: Overlap of tools detecting the *Time Dependency* vulnerability in source code.

## Appendix B.
## Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

### B.1. Summary

This paper provides a comprehensive analysis of existing vulnerability scanners for Ethereum-based smart contracts. In total, 17 vulnerability scanners are considered, both, bytecode and source code-based, that rely on detection methods of four categories: Static Analysis, Symbolic Execution, Fuzzing and Machine Learning. The study shows a high inequality in the identification of vulnerabilities, both for source code and byte code data. The potential reasons therefore are mentioned as non-uniform definitions of vulnerabilities, the use of different compiler versions, software aging, and changing coding styles. Also, the new variations of attacks where vulnerability scanners are not keeping up with.

### B.2. Scientific Contributions

- Independent Confirmation of Important Results with Limited Prior Research
- Provides a New Data Set For Public Use
- Provides a Valuable Step Forward in an Established Field

### B.3. Reasons for Acceptance

1) Timely topic and important problem: Despite advances made in the recent years, smart contract vulnerabilities continue to prevail.
2) Important findings: Identifying the reasons behind smart contract vulnerability detection tools being unable to consistently detect vulnerabilities can help developing better detection tools in the future work.
3) Large scale study: To date, this is the largest study of smart contract vulnerability detection tools in terms of analyzed tools, datasets, and evaluation methodologies.
4) Datasets and data labeling scripts will be open-sourced.

### B.4. Noteworthy Concerns

1) The paper lacks in-depth analysis of factors that contribute to the poor performance of tools. One could expect more insightful findings, such as the extent to which combining tools improves detection accuracy, the increase in the number of cases requiring manual analysis, or suggestions for optimizing tool combinations for specific vulnerabilities.

## Appendix C.
## Response to the Meta-Review

We recognize the importance of the concern raised by the reviewers. However, conducting a detailed analysis of the tools involved in the study would necessitate a significant amount of manual work, potentially extending over several months to years, which is not feasible.