TU WIEN Informatics

Reentrancy in Ethereum Smart Contracts

Methods and Tools for Detection

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Lisa Fürst, BSc
Matrikelnummer 11775842

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ass.Prof.in Dr.in Monika di Angelo
Mitwirkung: Ao.Univ.Prof. Dr. Gernot Salzer

Wien, 21. Jänner 2025

Lisa Fürst	Monika di Angelo

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Reentrancy in Ethereum Smart Contracts

Methods and Tools for Detection

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering and Internet Computing

by

Lisa Fürst, BSc
Registration Number 11775842

to the Faculty of Informatics

at the TU Wien

Advisor: Ass.Prof.in Dr.in Monika di Angelo
Assistance: Ao.Univ.Prof. Dr. Gernot Salzer

Vienna, 21st January, 2025

Lisa Fürst	Monika di Angelo

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Lisa Fürst, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe. Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel" habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 21. Jänner 2025

Lisa Fürst

Danksagung

Zuallererst danke an meine Eltern und meine Schwester – ohne euch wäre das alles nicht möglich gewesen.

Danke an Monika und Gernot für eure fachliche Unterstützung und eure Geduld bei der Betreuung dieser Arbeit. Monika, danke für dein schnelles Feedback und deine immer aufbauenden Worte, das hat mir unglaublich geholfen.

Acknowledgements

Firstly, thank you to my parents and my sister – without you, this wouldn't have been possible.

Thank you to my advisors, Monika and Gernot, for both your academic support and your incredible patience. Monika, thank you for your prompt and encouraging feedback, it meant a lot to me.

Kurzfassung

Obwohl die Softwareschwachstelle Reentrancy schon seit mehreren Jahren in der Ethereum-Community bekannt ist, stellt sie dennoch nach wie vor ein Problem dar, wenn es um die Sicherheitsanalyse von Smart Contracts geht. Aus diesem Grund gibt es regelmäßige Veröffentlichungen in diesem Forschungsgebiet, die darauf abzielen, verbesserte Methoden zur Verhinderung oder Erkennung von Reentrancy vorzustellen.

In dieser Arbeit versuchen wir, die bereits vorhandene Forschung in dem Gebiet zusammenzuführen und einen Überblick bereitzustellen, der die verschiedenen Arten von Reentrancy sowie die Code Patterns zur Vermeidung der Sicherheitslücke beschreibt, wobei der Fokus auf der Programmiersprache Solidity liegt. Weiters präsentieren wir verschiedene Ansätze zur Analyse von Smart Contracts und heben deren Stärken und Schwächen hervor. Eine Literatursuche, die auf einer vorhergehenden Arbeit basiert, wird genutzt, um die kürzlich entstandenen Publikationen in dem Bereich der Smart Contract-Analyse zusammenzutragen. Insgesamt identifizieren wir 104 neue Tools, die laut Angaben der Autoren geeignet sind, um Ethereum Smart Contracts auf Reentrancy zu untersuchen.

Zusätzlich wählen wir anhand der gesammelten Informationen vier Analysetools, sowohl neuere als auch etabliertere, und testen diese auf einem für diese Arbeit geschriebenen Datenset bestehend aus Smart Contracts. Das Datenset enthält alle vier Arten von Reentrancy, die in der Arbeit behandelt werden, und deckt die Solidity-Versionen 0.4 bis 0.8 ab. Die Daten sind öffentlich auf GitHub verfügbar und beinhalten sowohl Quellcode als auch Bytecode. Die Ergebnisse unseres Experiments zeigen, dass alle vier Tools – Oyente, Mythril, eThor und Sailfish – Probleme bei der Erkennung verschiedener Arten von Reentrancy haben, und dass ihre Leistung stark variiert, je nachdem, ob nach Reentrancy-Typ, Solidity-Version oder Codetyp beurteilt wird. Während drei von vier Tools in der Lage sind, die einfachste Form von Reentrancy konsistent zu erkennen, so werden die komplexeren Formen nur von jeweils einem der Tools entdeckt. Unsere Ergebnisse legen nahe, dass es innerhalb der Ethereum-Community noch an Bewusstsein für die komplexeren Arten von Reentrancy mangelt, und dass die Kombination mehrerer Analysetools sowie das gezielte Aussuchen eines Tools basierend auf den Eigenschaften des zu testenden Contracts die besten Chancen bieten sollten, um Smart Contracts frei von Reentrancy zu halten.

Abstract

Despite having been well known in the Ethereum community for many years, reentrancy still poses a problem when it comes to Smart Contract security analysis. For this reason, there are regular innovations in terms of newly proposed approaches to mitigate or detect this vulnerability.

In this paper, we attempt to combine the available research on the topic by providing an overview of the identified types of reentrancy and the coding patterns used to avoid them, focusing on the programming language Solidity. Furthermore, we present different Smart Contract analysis approaches and highlight their strengths and weaknesses. A literature review based on previous work in the field is used to collect recent research on the topic. In total, we identify 104 new tools that, according to the authors, are suitable for reentrancy detection in Ethereum Smart Contracts.

Furthermore, a combination of both recent and more established tools is selected and subsequently tested on a data set containing all four reentrancy types as well as five different Solidity versions, 0.4 to 0.8. The data set, which contains both source code as well as runtime bytecode, is built by hand specifically for this thesis and made available on GitHub. The results show that all four of the tools – Oyente, Mythril, eThor, and Sailfish – struggle with certain types of reentrancy and greatly vary in performance depending on whether the data is split by reentrancy type, Solidity version, or file type. While three out of four tools manage to consistently detect the simplest form of reentrancy, same-function reentrancy, the more complex types cross-function, delegated, and create-based reentrancy are detected by only one tool each. Our results suggest that there is a lack of awareness with regard to the more complex types of reentrancy within the tool development community, and that users wanting to keep their contracts secure might fare best by consulting multiple Smart Contract analysis tools or by tailoring their tool choice to the properties of the contract that is to be analyzed.

Contents

CHAPTER 1

Introduction

1.1 Problem statement and motivation

In recent years, there has been a rise in popularity of all things "Blockchain" – especially Bitcoin, a cryptocurrency that uses this technology. Introduced in 2008, it has since revolutionized decentralized currencies and has paved the way for many more blockchain-based currencies. Multiple cryptocurrencies have been created that allow scripts and applications to be created, now offering a number of oftentimes complex features that far surpass the limited possibilities of the early days [1]. Ethereum, which was introduced in 2014, is a prominent example for such a cryptocurrency platform and attracts the highest number of developers compared to its competitor networks [2]. It offers a Turing-complete language called "Solidity", which allows the creation of so-called "Smart Contracts". Smart Contracts are programs, ranging from simple constructs to complex applications for a range of use cases, that introduce new challenges for both users and developers. Rectifying erroneous code published to the blockchain may be hard or, in most cases, even impossible, thus requiring diligent work and adequate quality assurance measures. Missing bugs can easily turn out to be a costly mistake, as the best-known attack on a Smart Contract, named "DAO attack", illustrates. A vulnerability allowed an adversary to divert resources to their own accounts, resulting in the loss of more than 3.6 million ether, estimated to be worth around sixty million USD [3]. Caused by bad program design and characterized by allowing recursive calls to a contract, the vulnerability is now referred to as "reentrancy" and will be the focus of this thesis.

To help improve the quality of Smart Contracts published to the blockchain and increase security, several tools have been developed that allow checking Solidity Smart Contracts for a variety of vulnerabilities. However, a comparison of the results of such tools has revealed that there is low overlap between their reports, i.e. in many cases the tools do not agree on whether or not the analyzed contracts contain specific security concerns [4]. This

leads to the conclusion that detecting even well-known vulnerabilities such as reentrancy, which has been attempted using a variety of methods, is not a trivial task.

It is thus necessary to take a closer look at existing security analysis tools for Smart Contracts, as identifying differences and similarities may lead to a better understanding of what improves the performance of such tools, and may provide further insights for both Smart Contract and analysis tool developers.

1.2 Research questions

We aim to address the following research questions:

1. **What are the main aspects of reentrancy?** We research and compare existing definitions and provide examples of reentrancy in different contexts.

2. **Which approaches are used to detect reentrancy issues in Smart Contracts for Ethereum?** Various techniques have been proposed that are intended to uncover reentrancy issues in Smart Contracts, both using source code and bytecode. We research suitable categorizations for such approaches, highlighting differences and commonalities as well as their benefits and drawbacks.

3. **Which open-source tools address reentrancy in Smart Contracts and which approaches do they apply?** We collect currently available open-source security analysis tools that offer reentrancy detection.

4. **Which countermeasures are proposed?** To prevent reentrancy in otherwise vulnerable code, countermeasures may be introduced, for example in the form of specific programming patterns. We research and compare such measures.

5. **What are the discrepancies between the analysis results of the open-source tools and what are their causes?** Keeping in mind that previous evaluations have shown that tools may disagree on whether a Smart Contract contains a vulnerability or not, we evaluate whether this trend can be detected on the used data set. If so, we intend to analyze possible causes.

By answering these questions, we expect to provide a comprehensive definition of the vulnerability and its countermeasures, allowing readers to gain a deeper understanding of reentrancy and how to prevent it. The work further results in a comparison of analysis methods used for vulnerability detection and lists existing tools that use such methods. We conclude the thesis with both a quantitative and a qualitative assessment of the tools, providing deeper insights into their performance. Using our results, we hope to assist both developers and researchers in making informed decisions regarding the choice and further development of Smart Contract analysis tools.

1.3 Methodology

1.3.1 Literature review

An extensive literature review is necessary to determine the current state-of-the-art regarding techniques used by code analysis tools in the field of blockchains. In this context, scientific papers are reviewed to find concretely proposed code analysis tools.

1.3.2 Definition of selection criteria and tool selection

We define suitable selection criteria to ultimately select a smaller set of analysis tools that are reviewed and evaluated in the experimental study.

1.3.3 Extraction of methods from tools

By inspecting the tools' code as well as reading their corresponding publications we determine their analysis techniques and extract further information about their inner workings that may explain differences within the same techniques.

1.3.4 Construction of test data

For the experimental evaluation of the selected tools, a data set containing the corresponding ground truth is necessary. Such data is currently only available in small numbers, which is why we construct an additional benchmark set. The data consists of Solidity source code as well as the corresponding runtime bytecode, and contains labels that describe whether the Smart Contract is reentrant or not.

1.3.5 Experimental study

We execute the selected tools on the constructed data set to extract the corresponding analysis data.

1.3.6 Evaluation and comparison of approaches

The results of the experimental study in combination with the extracted methods serve as a basis for evaluating the tools' performance and identifying similarities and differences.

1.4 Structure of the thesis

The thesis is structured as follows. First, we present an overview of topics that are necessary to understand the context of the work and introduce blockchains and the Ethereum network. Chapter 3 then explains the intricacies of the reentrancy vulnerability, including how to avoid it, and provides code examples. The following chapter dives into the topic of Smart Contract analysis and provides an understanding on how to categorize analysis tools based on their used methods. Chapter 5 contains the results of

the aforementioned literature review and briefly presents the currently available Smart Contract analysis tools. In the next chapter, the setup of the experimental study is described and the capabilities of the selected tools are introduced. Chapter 7 provides the experiment's results and evaluates the tools based on both the results and the information gathered in the previous chapters. The last chapter then contains a summary of the thesis and provides a short outlook for possible future work on the topic.

CHAPTER 2

Background

This chapter provides an overview of the background information needed to understand the following chapters, and therefore focuses on important aspects of the cryptocurrency network Ethereum.

2.1 Blockchain and cryptocurrencies

Cryptocurrencies are peer-to-peer systems that rely on cryptography during their currency generation and distribution processes [5]. Transactions made using the currency are stored on a distributed public ledger, called blockchain, which provides integrity through the usage of hashes, thus guaranteeing data on the blockchain cannot be altered unnoticed. As there is no central entity responsible for verifying transactions, this is done through a verification process called "mining", which may also generate new currency units as an incentive to the people participating. To discourage malicious intent, mining is deliberately designed to be resource-intensive. Two popular strategies are called proof-of-work and proof-of-stake. Proof-of-work cryptocurrencies require miners to solve resource-intensive tasks, whereas in proof-of-stake systems validators need to prove they own a certain amount of the currency that can be used as collateral in the case of malicious intent. The cryptocurrency Bitcoin uses proof-of-work as a consensus mechanism [6], whereas Ethereum switched to proof-of-stake in 2022 [7].

2.2 Ethereum

Addressing several limitations of Bitcoin, in 2014 Vitalik Buterin published his ideas regarding a new cryptocurrency network in what is now known as the Ethereum whitepaper [8]. The intent behind Ethereum was to provide a built-in Turing-complete language to allow users to build a variety of systems and thus greatly increasing the possibilities provided by the network. Such systems are called "Smart Contracts" and allow creating

both simple programs as well as highly complex applications. Any transactions (e.g. transferring currency or executing a Smart Contract) made on the network require Ethereum's currency, called ether (ETH), to be paid.

Interactions in Ethereum are carried out between so-called accounts, which may fall into one of two categories: externally owned accounts and contract accounts. Externally owned accounts do not have any associated code, whereas contract accounts contain specific code, a Smart Contract, which is executed during interactions with other accounts. Each account has a unique 20-byte address used for communication between accounts. The state of each account consists of four fields:

- A transaction counter called "nonce",
- the amount of ether belonging to the account,
- the corresponding code, in case of a contract account,
- and the account's storage.

A contract account's storage may be written to through its code, while an externally owned account's storage must remain empty due to not having any associated code. Both externally owned accounts and contract accounts may send messages through the network, with contract accounts having the option to programmatically respond to received messages, thus allowing the concept of functions. A single transaction sent by an externally owned account may result in multiple messages being created. To prevent any code from running infinitely, each computation step required to perform the desired action needs to be paid for with ether; this fee is referred to as "gas".

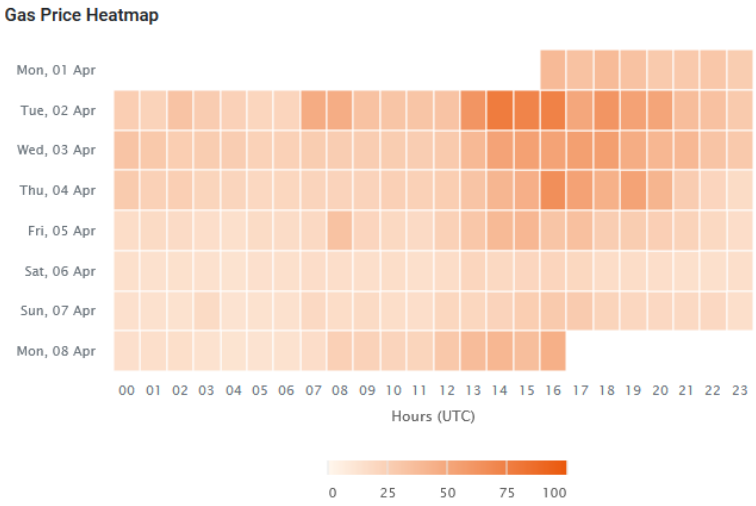

Figure 2.1: Gas price heatmap in gwei, April 2024 [9]

The sender of a transaction may specify the maximum amount of gas they are willing to pay (“start gas”), as well as the price per computation step (“gas price”). The gas price heavily influences the time it takes for a transaction to be processed, as gas fees are given to validators as an incentive, which makes it more likely for high-value transactions to be accepted sooner [10]. As can be seen in figure 2.1, the average gas price may fluctuate greatly even throughout the day. The fees are commonly expressed in “gwei”, with one gwei equating to one-billionth of an ether [11].

2.2.1 Ethereum Virtual Machine

An important foundation of Ethereum is the so-called Ethereum Virtual Machine (EVM), which allows the Ethereum network to act as a state machine and thus makes the execution of Smart Contracts possible [12]. The EVM can be divided into three parts: the machine state, the world state, and the virtual read-only memory (ROM). The world state describes an account’s persistent storage, including stored variables which may be manipulated via code execution. The virtual ROM contains the EVM code, i.e. the bytecode of the program to be executed. The EVM operates on a stack, which is part of the machine state. The machine state further contains the transaction-specific program counter, memory and gas.

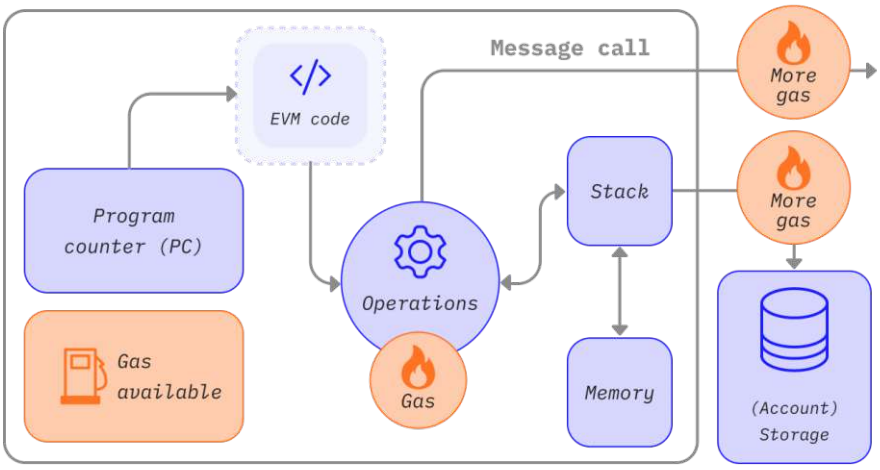

Figure 2.2: EVM program execution and gas usage [12]

Figure 2.2 provides an overview of the EVM’s functionality regarding program execution. The EVM code contains the operations to be completed, with each operation costing a certain amount of gas. Operations interact with the stack and make use of the volatile memory. Optional external calls and storage access may consume additional gas. The program execution is limited by the available gas; the program execution is stopped as soon as the available gas is exceeded.

When compiling a Smart Contract, a distinction is made between the “deployment”

and the "deployed" code [13]. The deployment code, also called creation code, includes additional initialization code and is run only once upon deploying the Smart Contract to the blockchain. It is mainly used for initializing the contract's storage and persisting the deployed code. Interactions with a deployed contract however trigger the execution of its deployed code, also called runtime code.

As mentioned, EVM code generally contains multiple operations which together make up a program. The EVM instruction set consists of all valid bytecode operations, referred to as opcodes, as described by the Ethereum yellow paper [14]; an excerpt is shown in table 2.1. As can be seen, besides basic arithmetic and bitwise operations, Ethereum-specific opcodes such as BALANCE or GASPRICE are offered. The number of operands which are removed from the stack and involved in the computation depends on the specific opcode; for example, an addition (0x01 ADD) removes two items from the stack and adds them together, while 0x60 PUSH1 does not remove any stack items but instead pushes one byte onto the stack.

Opcode	**Mnemonic**	**# operands**	**Description**
0x00	STOP	0	Halts execution
0x01	ADD	2	Addition operation
0x14	EQ	2	Equality comparison
0x16	AND	2	Bitwise AND operation
0x31	BALANCE	1	Get balance of given account
0x3a	GASPRICE	0	Get price of gas in current environment
0x52	MSTORE	2	Save word to memory
0x55	SSTORE	2	Save word to storage
0x60	PUSH1	0	Place 1 byte item on stack
0xf1	CALL	7	Message-call into an account

Table 2.1: EVM instruction set excerpt [14]

2.2.2 Smart Contract programming

Aligning with Ethereum's goal of providing a Turing-complete language which allows the creation of distributed applications using the blockchain, Solidity was first released in 2015 and has since become a popular programming language for the Ethereum network [15]. Starting with version 0.1.0, Solidity is in a state of constant development, with the latest version being 0.8.28 at the time of writing [16].

Solidity is influenced by multiple existing programming languages, most notably C++ and Python [17]. Listing 2.1 shows a simple Smart Contract written in Solidity. The `pragma` directive in line 1 describes the compiler versions that may be used when compiling the contract [18]. An error is generated should the compiler not match the directive. In the given example, `pragma solidity ^0.8.0`, any compiler version between 0.8.0 up to but not including 0.9.0 would be accepted.

A contract is defined using the keyword `contract` followed by the contract name, similar to the `class` keyword used in many object-oriented languages [19]. Contracts may contain state variables and functions and may also define structs and enums. Functions require a visibility modifier such as `public` or `private` and also allow additional modifiers, including custom modifiers defined in the contract. The `helloWorld()` function beginning in line 5 is a public function which does not read or modify the state (`pure`) and returns a non-persistent string (`string memory`).

Listing 2.1: "Hello World" example in Solidity v0.8

```
1 pragma solidity ^0.8.0;
2
3 contract HelloWorld {
4
5     function helloWorld() public pure returns (string memory) {
6         return "Hello World";
7     }
8 }
```

Due to breaking changes and bugfixes introduced with version upgrades, any Solidity Smart Contract analysis needs to consider the version's specific security implications. For example, while developers had to manually check for under- and overflows up until version 0.7, Solidity version 0.8.0 introduced automatic checks which avoid this type of bug [20].

Smart Contract developers are not required to use Solidity, as other Smart Contract programming languages for Ethereum do exist. One example is Vyper, which is heavily influenced by Python and prioritizes security and simplicity at the cost of offering fewer features than Solidity [21]. Due to Vyper's comparatively lower popularity, the following chapters will focus exclusively on Solidity.

CHAPTER 3

Reentrancy

3.1 Definition and categorization

Reentrancy, also written re-entrancy, is a programming pattern in which a contract A passes execution to another contract B, which in turn calls one of A's functions, thus re-entering contract A, all within a single transaction [22]. This behavior is most commonly the result of the fact that upon transferring ether or explicitly calling the `call()` function, the caller's fallback function is automatically invoked, in which the caller may execute arbitrary code. As reentrancy requires code execution, both caller and callee need to be contract accounts.

3.1.1 Reentrancy as a vulnerability

Figure 3.1 shows an example of an attack exploiting the reentrancy pattern. Whenever the attacker calls the victim's `withdraw()` function, the victim first checks whether the attacker has any stored funds, and if so, proceeds to transfer them to the attacker (`msg.sender.call()`). Besides transferring the ether to the attacker, at that point the attacker's `fallback()` function is automatically executed, in which the attacker proceeds to call the victim's `withdraw()` function again. As the victim has not yet updated the attacker's balance, they again send ether to the attacker and the cycle begins anew. This continues until the gas limit is reached.

While a reentrant contract might be vulnerable to attacks, this is not always the case. A vulnerability can be defined as "an error, flaw, or mistake in computer software that permits or causes an unintended behavior to occur" [23]. This means that a reentrant contract is considered non-vulnerable regarding reentrancy if the reentrancy is an intended feature and cannot be abused by an adversary. For example, a contract could contain the function `callCaller()` as shown in listing 3.1. Leaving aside the usefulness of such a contract, allowing users to repeatedly execute `callCaller()` using a fallback

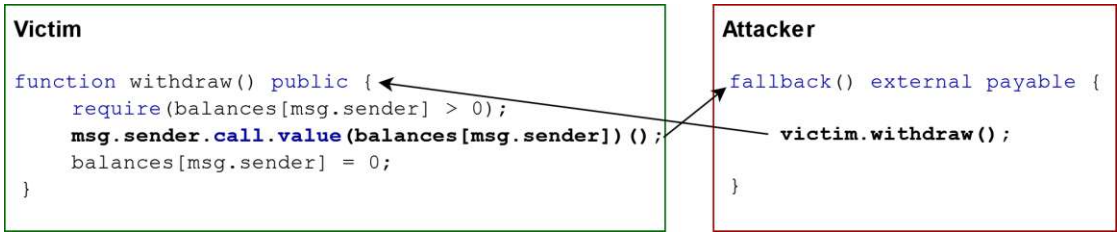

Figure 3.1: Sample Solidity code snippets showing a reentrancy attack

function does not automatically make it vulnerable, as there is no unexpected behavior that might be triggered in that case.

Listing 3.1: Non-vulnerable reentrant contract using a low-level call

```
1 pragma solidity ^0.8.0;
2
3 contract NonVulnerable {
4
5     function callCaller() public {
6         msg.sender.call("");
7     }
8
9 }
```

Despite the risks, it can be essential for Smart Contract developers to use functions that pass over the control flow to another contract. To avoid publishing code that contains an exploitable reentrancy, the Smart Contract Weakness Classification (SWC) registry recommends applying one of two best practices; firstly, the so-called Checks-Effects-Interactions pattern, and secondly, reentrancy locks [24].

When using the Checks-Effects-Interactions pattern, first, all requirements are verified (Checks), then all involved state variables are updated (Effects), and lastly, calls to other contracts may be performed (Interactions) [25]. Listing 3.2 shows how the vulnerability seen in figure 3.1 can be prevented using this pattern. By updating the balances (line 3) before calling the external contract (line 4), the attacker would fail the check `balances[msg.sender] > 0` upon re-entering the function, and no additional ether could be transferred.

Listing 3.2: Using the Checks-Effects-Interactions pattern to prevent a reentrancy vulnerability

```
1 function withdraw() public {
2     require(balances[msg.sender] > 0);
3     balances[msg.sender] = 0;
4     msg.sender.call.value(balances[msg.sender])();
5 }
```

Reentrancy locks, also called reentrancy guards, are function modifiers which allow a function to only be called once per transaction [26]. A simple example is shown in listing

3.3. The modifier `nonReentrant` first checks whether it has been called before, and if not, sets the internal lock variable `entered` to `true`. After that, the function which uses the modifier is run (line 6). After the function has been fully executed, the lock variable is reset to its original state. Should an attacker try to re-enter the function within the same transaction, the modifier's check would fail, as `entered` would still be set to `true` at that point.

Listing 3.3: Using a reentrancy guard to prevent a reentrancy vulnerability

```
1  bool internal entered = false;
2
3  modifier nonReentrant() {
4      require(!entered);
5      entered = true;
6      _;
7      entered = false;
8   }
9
10 function withdraw() public nonReentrant {
11     require(balances[msg.sender] > 0);
12     msg.sender.call.value(balances[msg.sender])();
13     balances[msg.sender] = 0;
14 }
```

In addition to the Checks-Effects-Interactions pattern and reentrancy guards, instead of transferring ether using the `call` function, `send` or `transfer` can be used to avoid a reentrancy vulnerability. While the lower-level `call` function forwards all available gas, both `send` and `transfer` automatically set a gas stipend of 2300 [27]. This means that the number of operations which can be executed inside the fallback function is severely limited, thus virtually preventing reentrancy, as the costs to re-enter would exceed the limit. However, it has to be noted that it is possible the functions' gas value is adapted in the future, which could lead to previously safe contracts to be then susceptible to attacks. For this reason, articles have been published which instead recommend using a combination of `call` and one of the aforementioned best practices (e.g. [28]).

3.1.2 Categorization

Based on the various manifestations of reentrancy as described in the previous subsection, we propose a categorization that allows to distinguish Smart Contracts based on their reentrancy-related properties. An overview can be seen in figure 3.2.

The categorization shows that Smart Contracts can be divided into two main groups, either containing code that allows re-entering the contract or not. Within the two groups, the contracts may be differentiated depending on additional properties. Additionally, we offer a separate category for contracts that are not currently reentrant but are at risk of becoming vulnerable should there be adjustments regarding gas values in the future.

The non-reentrant contracts are evaluated according to whether they are non-reentrant due to the security measures used, or whether they contain no external calls that could be

exploited by an attacker. Reentrant contracts may either be vulnerable or non-vulnerable. In the case of a vulnerability, there is a category for contracts that allow an adversary to access the contract's ether balance, as opposed to allowing them to manipulate the contract's state in some other way.

3.2 Types of reentrancy

While examples for reentrancy vulnerabilities typically present Smart Contracts similar to the code shown in figure 3.1, this is only one of multiple different types of reentrancy. In addition to *same-function reentrancy*, Rodler et al. identify three further patterns which all result in a contract being re-entered: *cross-function reentrancy*, *delegated reentrancy* and *create-based reentrancy* [22].

The previously discussed same-function reentrancy allows re-entering the function that makes the external call, thus executing the same function at least twice. In cross-function reentrancy, on the other hand, a different function in the same contract is entered by the attacker, which may also lead to inconsistent states. Listing 3.4 shows a vulnerable contract. By calling `withdrawAll()` and then executing `transfer()` from the attacker's fallback function, the attacker can move tokens to a different account (line 22) even though their value has been paid out in full (lines 13-14).

Listing 3.4: Cross-function reentrancy example [29]

```
1  pragma solidity ^0.8.0;
2
3  contract Reentrancy {
4      mapping(address => uint256) tokenBalance;
5      mapping(address => uint256) etherBalance;
6
7      constructor() payable {}
8
9      function withdrawAll() public {
10         uint256 etherAmount = etherBalance[msg.sender];
11         uint256 tokenAmount = tokenBalance[msg.sender];
12         if (etherAmount > 0 && tokenAmount > 0) {
13             uint256 e = etherAmount + (tokenAmount * 2);
14             etherBalance[msg.sender] = 0;
15             msg.sender.call{value: e}("");
16             tokenBalance[msg.sender] = 0;
17         }
18     }
19
20     function transfer(address to, uint256 amount) public {
21         if (tokenBalance[msg.sender] >= amount) {
22             tokenBalance[to] += amount;
23             tokenBalance[msg.sender] -= amount;
24         }
25     }
26 }
```

Figure 3.2: Categorization of Smart Contract based on reentrancy-related properties

Note that this contract is not vulnerable to same-function reentrancy, as for the call to be executed, both `etherBalance` and `tokenBalance` need to be greater than zero (line 12). However, the ether balance is updated prior to the call, meaning that upon re-entering, the check would fail and the contract's state would stay consistent.

The next reentrancy type, delegated reentrancy, makes use of the `DELEGATECALL` or `CALLCODE` opcodes to hide an unsafe `call`. When a contract A uses these instructions to call another contract B, B can temporarily execute arbitrary code within the context of contract A, i.e. B may also update A's state. This is a useful functionality for implementing library contracts, however passing control to an unknown party can also introduce new weaknesses.

Listing 3.5 shows how Solidity's `delegatecall()` function can result in a reentrancy vulnerability. Calling the `withdraw()` function triggers the internal `_libsend()` function (line 25), which uses a delegatecall to pass control to an external library's code in order to transfer ether. While the `SafeSending` library does indeed transfer the ether, it does so with the unsafe `call()` function. As it runs within the context of the calling function, this makes the original contract vulnerable to a reentrancy attack, since the external call is executed before the internal state is updated (line 26).

Listing 3.5: Delegated reentrancy example [29]

```
1  pragma solidity ^0.8.0;
2
3  library SafeSending {
4      function send(address to, uint256 amount) public {
5          to.call{value: amount}("");
6      }
7  }
8
9  contract Bank {
10     mapping(address => uint) public balances;
11     address owner;
12     address safesender;
13
14     constructor(address _safesender) payable {
15         owner = msg.sender;
16         safesender = _safesender;
17     }
18
19     function donate(address to) payable public {
20         balances[to] += msg.value;
21     }
22
23     function withdraw(uint amount) public {
24         if (balances[msg.sender] >= amount) {
25             _libsend(msg.sender, amount);
26             balances[msg.sender] -= amount;
27         }
28     }
29
```

```
30      function _libsend(address to, uint256 amount) internal {
31          address(safesender).delegatecall(abi.encodeWithSignature("send(
        address,uint256)", to, amount));
32      }
33  }
```

The last type of reentrancy as presented by Rodler et al. [22] is called create-based reentrancy. In Ethereum, both externally-owned accounts and contract accounts may create new contracts using the EVM `CREATE` opcode. This triggers the execution of the newly created contract's constructor. While the new contract's code is generally known and trusted, it may contain further external calls which are not guaranteed to be safe. Should the newly created contract pass control to a malicious contract, this again may result in a reentrancy attack to the original contract, should the contract not be protected using one of the aforementioned measures.

Listing 3.6 illustrates a multi-contract setup where this is the case. Here, the original contract, Bank, triggers the creation of a new contract, Intermediary, within the `withdraw()` function (line 35). Note that again, the contract's state is updated in line 36, after the potential external call caused by line 35. During the contract creation process, the constructor of Intermediary does indeed call an external contract in line 17, an instance of IntermediaryCallback, whose address equals the caller of Bank's `withdraw()` function and thus cannot be known in advance by the Bank contract, making it potentially unsafe.

An attacker can exploit Bank's vulnerability by first depositing some ether into the Bank contract before calling its `withdraw()` function. During the creation of the Intermediary contract, the control is passed back to the attacker, who may now call `withdraw()` again, triggering the creation of another Intermediary contract. After that, the attacker may decide to allow the previously called `withdraw()` functions to resume normally instead of repeatedly calling them. As the `withdraw()` function was previously called twice, this means that the balance is set to zero twice (line 37), while it is also transferred twice (lines 38-39), once per created Intermediary contract. As the attacker has control over the Intermediary contracts due to being set as their owner, they may now withdraw the stored ether using the `transferEther()` function, resulting in the attacker gaining twice as much ether as intended by the Bank contract.

Listing 3.6: Create-based reentrancy example (cf. [29])

```
1   pragma solidity ^0.8.0;
2
3   abstract contract IntermediaryCallback {
4       function registerIntermediary(address what) public virtual payable;
5   }
6
7   contract Intermediary {
8       address owner;
9       Bank bank;
```

```
10      uint amount;
11
12      constructor(Bank _bank, address _owner, uint _amount) {
13          bank = _bank;
14          owner = _owner;
15          amount = _amount;
16
17          IntermediaryCallback(_owner).registerIntermediary(address(this));
18      }
19
20      function transferEther() public {
21          if (msg.sender == owner) {
22              payable(msg.sender).transfer(amount);
23          }
24      }
25
26      fallback() external payable {}
27  }
28
29  contract Bank {
30      mapping(address => uint) balances;
31      mapping(address => Intermediary) subs;
32
33      function withdraw() public {
34          if (balances[msg.sender] > 0) {
35              uint amount = balances[msg.sender];
36              subs[msg.sender] = new Intermediary(this, msg.sender, amount);
37              balances[msg.sender] = 0;
38              payable(address(subs[msg.sender]))
39                  .transfer(amount);
40          }
41      }
42
43      function deposit() public payable {
44          balances[msg.sender] += msg.value;
45      }
46  }
```

CHAPTER 4

Smart Contract Analysis

Code analysis is a useful approach to reduce the possibility of releasing code containing unwanted behaviors, e.g. vulnerabilities such as reentrancy. For this reason, many analysis techniques have evolved, which are used to preemptively scan programs such as Smart Contracts for security issues. Analysis tools can be divided into static and dynamic analysis, as shown in figure 4.1. In their 2020 survey, Kim and Ryu [30] identify multiple subtypes of static and dynamic analysis. This chapter discusses these types and presents examples of their usage in various Smart Contract analysis tools.

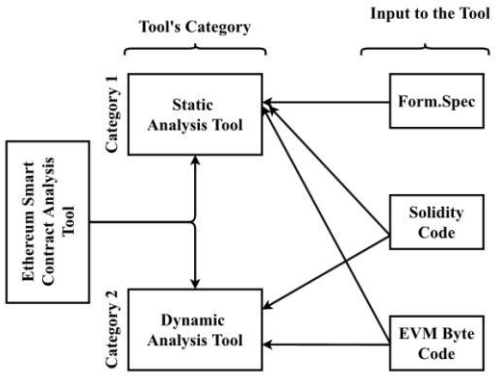

Figure 4.1: Categorization of Ethereum Smart Contract analysis tools based on analysis type [31]

4.1 Static analysis

Static code analysis is static in the sense that the program does not need to be executed, but instead only the program's source or bytecode is required to perform the analysis.

4.1.1 Symbolic execution

In symbolic execution, a concrete execution of the code is simulated using symbolic value inputs [30]. A symbolic value represents an arbitrary value which adheres to program-specific constraints, which allows exploring multiple execution paths instead of just one, as would be the case with concrete values [32]. The advantages of symbolic execution can be illustrated using listing 4.1. The simple function `foobar()` takes two input values `a` and `b` and concludes with an assertion. Ideally, the assertion should never fail, which can be tested using a variety of techniques. For example, fuzzing is a popular approach in which the function would be executed using a multitude of randomly selected values for `a` and `b`. However, it is possible that inputs triggering the assertion are not found using this approach, as the input values can each be chosen from a range of 2^{32} values, assuming an `int` is stored as a 4-byte value. By using symbolic values instead of concrete values, the obstacle of having to rely on randomly generated values can be overcome.

Listing 4.1: Symbolic execution example: Which input values cause the assertion to fail? [32]

```
1 void foobar(int a, int b) {
2     int x = 1, y = 0;
3     if (a != 0) {
4         y = 3 + x;
5         if (b == 0)
6             x = 2 * (a+b);
7     }
8     assert(x - y != 0);
9 }
```

To solve the problem using symbolic execution, a tree is built, representing the execution paths of the program. Each program path is evaluated, using symbolic values where the actual values are not known, thus gradually refining the conditions the symbolic values need to meet to execute specific program paths. The tree for listing 4.1 is depicted in figure 4.2. Each point in the program is represented by the tuple $(stmt, \sigma, \pi)$, with $stmt$ being the next statement that is evaluated, σ being the so-called symbolic store, and π being the path constraints. The symbolic store σ contains all known variables, either symbolic or concrete, while the path constraints π describe the constraints posed on the symbolic values in order to reach $stmt$.

As an example, after evaluating line 2, the symbolic store σ consists of the values $\{a, b, x, y\}$, with a and b being assigned the symbolic values α_a and α_b, x being assigned 1 and y being assigned 0. The path constraint π equals $true$, as there are no requirements that need to be met in order to reach this line of code. The next statement to be evaluated, $stmt$, equals line 3, i.e. `if (a != 0)`. As this if-statement can evaluate to either `true` or `false`, the tree splits into two branches at this point, one branch adding the condition $\alpha_a = 0$ and the other $\alpha_a \neq 0$.
This process is repeated until every branch reaches the end of the program, which in

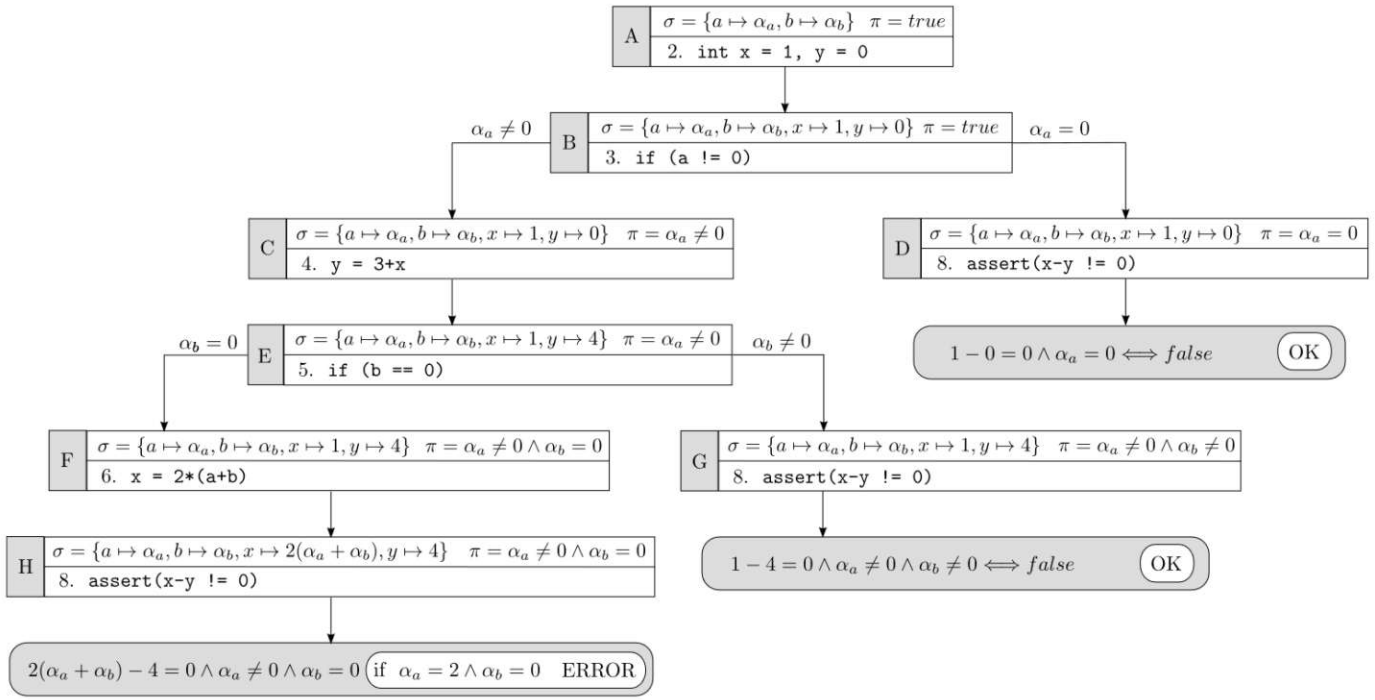

Figure 4.2: Symbolic execution tree of the function given in listing 4.1 [32]

this case is the assert-statement `assert(x - y != 0)`. Using each of the branches' symbolic store and path constraints, the statement is then evaluated. In this example, we finally arrive at a branch which contains the symbolic store $\sigma = \{a \mapsto \alpha_a, b \mapsto \alpha_b, x \mapsto 2(\alpha_a + \alpha_b), y \mapsto 4\}$ as well as the path constraints $\alpha_a \neq 0$ and $\alpha_b = 0$. By inserting the symbolic store's values into the assert-statement, we receive $2(\alpha_a + \alpha_b) - 4 \neq 0$. Due to the path constraints we know that α_b equals zero, while α_a is a number other than zero. Thus, we can simplify the equation to $2\alpha_a - 4 \neq 0$. We can now easily see that if α_a is assigned 2, the equation does not hold and the assertion therefore fails.

As symbolic execution programs are expected to provide the results to such equations without having to rely on users to solve them, they may integrate so-called SMT solvers for this task, for example Z3 [33] or cvc5 [34].

Baldoni et al. [32] identify four main challenges in symbolic execution. Firstly, managing code that relies on data stored in memory, such as pointers, may require addresses to be treated as symbolic values as well and thus increases the difficulty. Another big challenge is the occurrence of external calls in the analyzed code – it may be infeasible to consider every possible outcome and side effect in such cases. Thirdly, in more complex programs, having to evaluate every single possible path may lead to state space explosion and thus would require the analysis to run for an extended amount of time. The fourth challenge is the previously mentioned need to use solvers as part of the analysis process, which may be inefficient in the case of complex constraints.

Some examples for Smart Contract analysis tools that use this approach are Oyente [35], Mythril [36], and Conkas [37].

4.1.2 Abstract interpretation

The main idea behind abstract interpretation is to simplify concrete computations by mapping the operations to an abstract universe, which can then help evaluate certain properties of the program, for example partial correctness. Cousot and Cousot [38] explain this using a mathematical example which uses the rule of signs, i.e. the behavior of the plus and minus sign in operations such as addition and multiplication. The abstract universe, in this case, is $\{(+), (-), (\pm)\}$, and any mathematical expressions are expressed within this universe. For example, the equation $-1515 * 17$ would be described as $-(+) * (+) \Rightarrow (-) * (+) \Rightarrow (-)$, which allows to conclude that the solution of the problem must be a negative number, without having to calculate the actual expression. Abstract interpretation, however, may also cause imprecise or uncertain results, as the expression $-1515 + 17$ shows. It is mapped to $-(+) + (+) \Rightarrow (-) + (+) \Rightarrow (\pm)$, as the result's sign of an addition between a negative and a positive number may be either negative or positive, depending on the concrete values.

Multiple tools use this approach to analyze EVM bytecode for vulnerabilities, for example MadMax [39] and Vandal [40].

4.1.3 Machine Learning techniques

With the rising popularity of Artificial Intelligence (AI), Machine Learning (ML) models have emerged in the field of Smart Contract analysis. Many approaches apply a pre-processing step in which the code is represented as a graph, for example Control Flow Graphs (CFG) or Abstract Syntax Trees (AST) [41]. Furthermore, different types of ML models may be used, from highly complex neural networks to simpler Support Vector Machines (SVM).

Graph Neural Networks (GNN) are especially suited for the analysis of graph-like data, which is why Cai et al. [42] apply this type of network on Solidity representations which are combinations of CFGs, ASTs and Program Dependency Graphs (PDG). After the code is transformed, the resulting graph is reduced in a so-called program slicing step, therefore removing nodes which are irrelevant for vulnerability detection. The graph is then fed to a previously trained GNN, which outputs whether the code is likely to include a vulnerability or not. Cai et al. test their model on a combination of four existing Smart Contract databases, containing 7000 vulnerable contracts and 6779 contracts without vulnerabilities. They report that their approach results in an accuracy of 90.2%, compared to an accuracy of 51.1% using the tool Mythril.

Tann et al. [43] propose using a so-called long short-term memory (LSTM) model, i.e. a neural network, to classify an Ethereum smart contract's opcodes into "vulnerable" or "not vulnerable", as can be seen in figure 4.3. Their model is trained on a total of 620,000 smart contracts, with their classification labels being provided by the tool Maian [44] and a number of false positives being removed as suggested by the tool's developers. Using this approach, they achieve an accuracy of over 99% on a test set consisting of 120,000 contracts. However, as the neural network is trained on a sequence of opcodes, it may struggle with detecting vulnerabilities based on control-flow or data-flow properties.

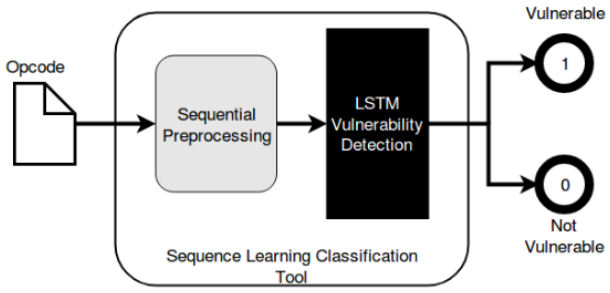

Figure 4.3: Machine Learning-based Smart Contract vulnerability analysis tool [43]

Huang [45] presents a solution based on a Convolutional Neural Network (CNN), in which a Solidity contract's bytecode is represented as a fixed-size RGB image and then analyzed by the model in order to detect compiler bugs. They test different parameters during the training process and report an accuracy of 83% to 96% on a test set collected

from etherscan.io.

4.2 Dynamic analysis

Dynamic analysis can be defined as "the analysis of the properties of a running program" [46]. This means that programs can be analyzed dynamically even if their source code or bytecode is not available, as dynamic analysis generally does not require any knowledge of the inner workings of a program.

4.2.1 Fuzzing

Fuzzing involves using test data, either randomly generated or according to predefined rules, as input for a program. The hope is to find inputs that trigger undesirable outcomes and thus reveal bugs in the software [47]. While fuzzing can be carried out using a purely dynamic approach using random input data, there are benefits to combining it with a static analysis, as it can help reduce the number of test inputs that need to be generated in order to trigger specific conditions. To detect assertion violations and memory-access errors, the Smart Contract analysis tool *Harvey* [48] generates a test suite that aims to cover all program paths by keeping track of which paths have already been visited. Instead of randomly mutating inputs, as similar fuzzers do, Harvey uses an input prediction algorithm to increase the chances of generating inputs for paths that have not been visited. Additionally, Harvey fuzzes transaction sequences, again keeping track of code coverage, therefore avoiding the creation of unnecessarily long sequences that do not cover new paths. Liu et al. [49] attempt to improve this approach by additionally taking into account the different states of a contract. By evaluating the data dependencies of the contract, i.e. variables that may be written or read within different functions, the transaction sequence generation may be improved. They additionally include a branch search algorithm which targets branches that are more likely to contain vulnerable code, thus using more fuzzing resources on them and increasing the chances of finding the correct input to trigger a vulnerability. Liu et al. report that using this approach, their fuzzer is able to detect reentrancy vulnerabilities.

4.2.2 Runtime verification

Runtime verification is used to check certain correctness properties of a program, i.e. in this case a Smart Contract, during runtime [50]. To conduct runtime verification, both the program to be monitored and the property are used as inputs. The properties may be specified using various formalisms, but can also take other forms depending on the verification tool, for example properties in the form of code written in certain programming languages may also be accepted. In a runtime verification approach by Grossman et al. [51], the authors define the property Effectively Callback Free (ECF), which ensures that a Smart Contract behaves as though no callbacks occur during its execution, which is relevant for detecting reentrancy vulnerabilities. The execution traces are monitored and an Invocation Order Constraint graph is built, which is then checked

for any cycles. As cycles would indicate that the contract does not satisfy the ECF property, appropriate alerts can be raised in this case. *Sereum* [22] is another tool that uses runtime monitoring and analysis to detect reentrancy vulnerabilities in Smart Contracts. By checking the programs for storage variable dependencies and possible state inconsistencies, it allows detecting attacks using callbacks. The authors report a false-positive rate of 0.06% as well as a low runtime overhead.

4.2.3 Concolic testing

Concolic testing combines the concrete execution of a program with symbolic execution [52]. By doing so, limitations of both concrete and symbolic execution can be overcome – symbolic values help improve the test coverage by allowing to create test inputs that are guaranteed to cover various conditions, while randomly generated concrete inputs do not suffer from possible state space explosion and thus improve efficiency. Mossberg et al. [53] call their tool *Manticore* a "dynamic symbolic execution framework", another description for concolic testing. Manticore creates path predicates, i.e. input constraints, on paths that are part of its symbolic execution analysis, and uses these predicates to create targeted input data that is guaranteed to execute visited branches. This allows reproducible results, as any vulnerabilities found can be triggered during concrete execution using the inputs generated by the tool. Manticore works on Ethereum, Linux ELF, and WebAssembly (Wasm) bytecode. Marques et al. [54] attempt to improve the analysis of Wasm code and present WASP, a concolic execution engine, which on average achieves a 15.8x speedup compared to Manticore. This speedup may be caused by WASP's underlying faster programming language and its decreased memory usage.

4.2.4 Mutation testing

Unlike the previous methods, mutation testing is used to analyze the quality of a test suite [55]. By deliberately adding small changes to the source code of a Smart Contract, it is tested whether the applied tests are able to detect the now faulty code parts. Modified versions of the code are called mutants, and whether they are detected or not during the testing process determines if they are considered "killed" or "live" mutants. Analyzing the live mutants, i.e. the changes that are not caught by the test suite, can help improve the quality of the testing.
Barboni et al. [55] present a mutation testing tool called *SuMo*, which operates on Solidity code. By tailoring the mutation operators to the programming language, more meaningful mutations can be applied. Additionally, they attempt to decrease the creation of non-compilable mutants, which severely impact the tool's performance. They report that multiple real-world Ethereum applications that were shipped with test suites receive a low mutation score when using SuMo, meaning that the tests detect a low number of the inserted code changes. This suggests that using mutation testing optimized for Solidity may greatly improve the quality of testing suites and thus also the published contracts.

CHAPTER 5

Tools for Automated Reentrancy Detection

5.1 Literature review

5.1.1 Search strategy

The aim of the literature review is to extend Rameder's 2021 "Systematic Review of Ethereum Smart Contract Security Vulnerabilities, Analysis Methods and Tools" [56] by reentrancy detection tools that were published since Rameder's review was conducted. To allow for better comparability, we choose to conduct the review using the same online sources and keywords as Rameder, only narrowing down the search to include the keyword *reentrancy* as well. Thus, the primary search terms that are used in the review are *smart contract*, *security*, *vulnerability*, *bug*, *Ethereum*, *analysis*, *detection*, *verification*, *tool* and *reentrancy*, as well as their plural, where appropriate. The following five electronic sources are selected for the literature review:

- ACM Digital Library (`https://dl.acm.org`)
- Google Scholar (`https://scholar.google.com`)
- IEEE Xplore (`https://ieeexplore.ieee.org`)
- ScienceDirect (`https://www.sciencedirect.com`)
- TU Wien CatalogPlus (`https://catalogplus.tuwien.at`)

We choose to apply the same **exclusion criteria** as Rameder, only updating the criterion concerned with the recency of the literature:

- The work is not written in English
- The work is a patent, blog entry or other gray literature
- The work is not accessible online
- The full text of the work is not available for downloading via the TU Wien network
- The work is published before 2021

However, as we are only interested in reentrancy detection tools and want to exclude literature not presenting a novel tool or method, we do not apply Rameder's suggestions but instead select the following **inclusion criteria**:

- The contents make it clear that the work is related to a novel Smart Contract vulnerability detection tool or method
- The contents make it clear that the corresponding tool or method is suitable for reentrancy detection
- The contents make it clear that the work is either network-independent or suitable for Ethereum

5.1.2 Initial search results

The initial search was conducted on October 17, 2024 (ACM Digital Library) as well as October 19, 2024. The search queries used for each of the aforementioned search engines are presented in table 5.1 and are deliberately kept as similar as possible to Rameder's original search terms, with the exception that the word *reentrancy* is now a required keyword. In all searches, additional filters were applied to limit the publication dates to only include works from January 1, 2021 to the day of the search, i.e. October 2024.

The outcome of the initial search is shown in figure 5.1. The ACM Digital Library produces the highest number of results (103), followed by ScienceDirect (90), IEEE Xplore (57) and Google Scholar (22). TU Wien CatalogPlus produces the lowest number of results with only 9 papers in the result set. In total, we receive 281 pieces of literature.

5.1.3 Selection process

In the first step, the results are merged and duplicates removed. This brings the total number of results down to 256. Applying the exclusion criteria does not remove any further results, as all 256 works adhere to the criteria.

As the inclusion criteria aim at removing any papers that do not provide a method for reentrancy detection, they are applied by selecting only papers that explicitly mention their corresponding tool's or method's suitability for detecting reentrancy. This also means that tools focusing on problem solutions other than vulnerability detection, e.g.

Search engine	Search query
ACM Digital Library	Title:(("smart contract" OR "smart contracts")) AND All-Field:(reentrancy AND (vulnerability OR vulnerabilities OR bug OR bugs OR tool OR security OR analysis OR detection OR verification))
Google Scholar	allintitle: reentrancy ("smart contract" OR "smart contracts" OR Ethereum) vulnerability OR vulnerabilities OR bug OR bugs OR security OR analysis OR detection OR tool OR verification
IEEE Xplore	("All Metadata":smart contract OR "All Metadata":smart contracts) AND ("All Metadata":vulnerability OR "All Metadata":vulnerabilities OR "All Metadata":bug OR "All Metadata":bugs OR "All Metadata":tool) AND ("All Metadata":security OR "All Metadata":analysis OR "All Metadata":detection OR "All Metadata":verification) AND ("All Metadata":reentrancy)
ScienceDirect	Title, abstract, keywords: "smart contract" AND (vulnerability OR vulnerabilities OR bug OR tool OR security OR analysis OR detection) Find articles with these terms: reentrancy
TU Wien CatalogPlus	title contains (“smart contract” OR “smart contracts”) AND Subject contains reentrancy AND (security OR vulnerability OR vulnerabilities OR bug OR bugs OR analysis OR detection OR tool OR verification)

Table 5.1: Search queries used to retrieve the initial search results

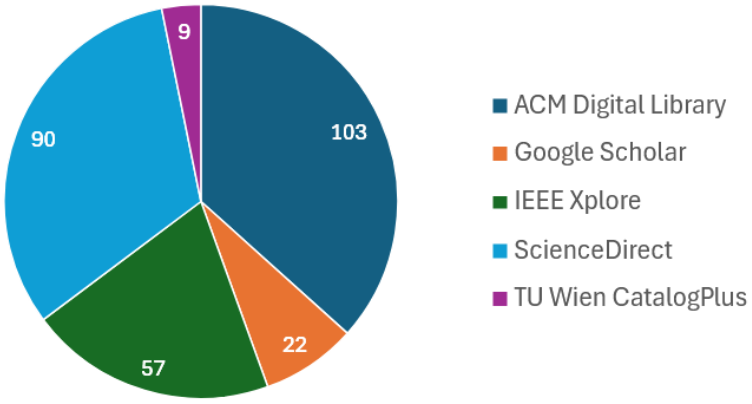

Figure 5.1: Initial search results based on the queries presented in table 5.1

preventing or repairing vulnerable contracts, are not included in the final selection.

Furthermore, mutation testing tools, which aim at improving test suites, are not covered by the inclusion criteria and thus removed. Of the 256 works in the first result set, 152 do not adhere to the inclusion criteria, resulting in a final count of 104.

5.2 Tool overview

An overview of all 104 papers describing Smart Contract analysis tools for reentrancy are is shown in table 5.2. As can be seen in 5.2, the majority of papers are part of conference proceedings or workshops, followed by journal articles and a single thesis paper. Out of all 104 results, 83 describe tools that use static analysis, while only 20 employ dynamic analysis. The thesis paper recommends an approach that combines both static and dynamic analysis. We were able to retrieve online resources containing the tools' source code for 33 papers. Two separate papers by Ren et al. ([57, 58]) describe the development of the same tool, *SCStudio*, resulting in a total of 103 unique tools.

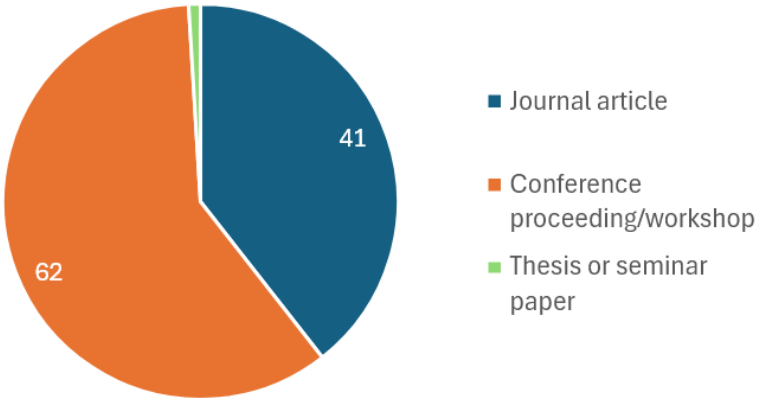

Figure 5.2: Publication types of literature selection

5.3 Static reentrancy detection tools

As the experimental part of the thesis focuses on static analysis tools using a traditional approach, i.e. Machine Learning solutions are not considered, the tools gathered in the literature review that fit this criteria are briefly presented in this section. We additionally include information about the tools *Oyente*, *Mythril*, *Slither*, and *Vandal*, as they are repeatedly used as baseline when evaluating new approaches. Furthermore, we describe *eThor*, which was developed at TU Wien. The literature is presented in alphabetical order.

Table 5.2: Literature review tool overview

Author	Year	Tool/method/model name	Publication type	Analysis type	Available?	Notes
Ali Khan and Siami Namin [59]	2024	TechyTech	Journal article	Dynamic	No	
Alkhalifah et al. [60]	2021	N/A	Journal article	Dynamic	No	https://github.com/Smart-Contracts-Verification/A-Verification-Approach-for-Composite-Smart-Contracts-Security-using-FSM
Almakhour et al. [61]	2023	N/A	Journal article	Static	Yes	https://github.com/ucsb-seclab/sailfish
Ashizawa et al. [62]	2021	Eth2Vec	Conference proceeding/workshop	Static	Yes	https://github.com/ls-osaka/eth2vec
Boi et al. [63]	2024	N/A	Journal article	Static	No	
Boi et al. [64]	2024	Vuln-HuntGPT	Conference proceeding/workshop	Static	No	
Bose et al. [65]	2022	SAILFISH	Conference proceeding/workshop	Static	Yes	https://github.com/ucsb-seclab/sailfish
Braghin et al. [66]	2024	N/A	Conference proceeding/workshop	Static	No	https://github.com/smart-contract-verification/ethereum-vis-asm
Cai et al. [42]	2023	N/A	Journal article	Static	No	
Cai et al. [67]	2024	N/A	Journal article	Static	No	
Cao et al. [68]	2021	ACID	Journal article	Dynamic	No	Intrusion detection system
Chen et al. [69]	2024	Clear	Conference proceeding/workshop	Static	Yes	https://github.com/cienpp1881/Clear
Chen et al. [70]	2024	SOChecker	Conference proceeding/workshop	Static	Yes	https://github.com/BugmakerCC/SOChecker
Chen et al. [71]	2023	EA-RGCN	Journal article	Static	Yes	https://github.com/frankdadale/EA-RGCN
Cheng et al. [72]	2024	VDCEP	Journal article	Static	Yes	https://github.com/Jamesken23/VDCEP
Cheng et al. [73]	2024	EGFL	Journal article	Static	Yes	https://github.com/Jamesken23/EGFL
Choi et al. [74]	2022	SMARTIAN	Conference proceeding/workshop	Dynamic	Yes	https://github.com/SoftSec-KAIST/Smartian both static and dynamic analysis
Demir et al. [75]	2023	N/A	Conference proceeding/workshop	Static	Yes	https://github.com/miralabai/blockchain-vulnerability-detection
Dong and Li [76]	2023	N/A	Conference proceeding/workshop	Static	No	
Duan et al. [77]	2023	N/A	Conference proceeding/workshop	Static	No	
Eshghie et al. [78]	2021	Dynamit	Conference proceeding/workshop	Dynamic	Yes	https://github.com/mojtaba-eshghie/Dynamit
Fang et al. [79]	2021	Jyane	Conference proceeding/workshop	Dynamic	No	
Feng et al. [80]	2023	SETT	Conference proceeding/workshop	Static	No	
Feng et al. [81]	2024	N/A	Journal article	Static	No	
Feng et al. [82]	2024	N/A	Journal article	Static	No	
Gao et al. [83]	2024	sGuard+	Journal article	Static	No	
Garfatta et al. [84]	2022	N/A	Conference proceeding/workshop	Static	No	
Gong et al. [85]	2023	GRATDet	Journal article	Static	No	
Guo et al. [86]	2022	SCVSN	Journal article	Static	No	
Guo et al. [87]	2023	N/A	Conference proceeding/workshop	Static	No	
Guo et al. [88]	2024	N/A	Journal article	Static	No	
HajiHosseinKhani et al. [89]	2024	SCsVulLyzer	Journal article	Static	Yes	https://github.com/ahlashkari/SCsVulLyzer
Han [90]	2024	N/A	Conference proceeding/workshop	Static	No	
He et al. [91]	2023	N/A	Journal article	Static	No	
He et al. [92]	2024	N/A	Journal article	Static	No	

Huang et al. [93]	2024	CSCO	Journal article	Static	No	
Jain and Tripathi [94]	2023	N/A	Conference proceeding/workshop	Static	No	
Lakadawala et al. [95]	2024	HARDEN	Conference proceeding/workshop	Static	No	
Le Hong et al. [96]	2023	N/A	Conference proceeding/workshop	Static	No	
Li et al. [97]	2022	ReDefender	Journal article	Dynamic	No	
Li et al. [98]	2023	N/A	Conference proceeding/workshop	Static	No	
Li et al. [99]	2023	N/A	Conference proceeding/workshop	Static	No	
Li et al. [100]	2024	VDM-AEI	Journal article	Static	No	
Li et al. [101]	2024	EvoFuzzer	Journal article	Dynamic	No	
Liu et al. [102]	2023	CGE, GPSCVulDetector	Journal article	Static	Yes	https://github.com/Messi-Q/GPSCVulDetector
Lin et al. [103]	2023	MRN-GCN	Journal article	Static	No	
Luo and Luo [104]	2023	VULDET	Conference proceeding/workshop	Static	No	
Luo et al. [105]	2023	TSCSG	Conference proceeding/workshop	Static	No	
Luo et al. [106]	2024	SCVHunter	Conference proceeding/workshop	Static	No	
Mittal et al. [107]	2023	SC-Defender	Conference proceeding/workshop	Static	No	
Narayana and Sathiyamurthy [108]	2023	N/A	Conference proceeding/workshop	Static	No	
Ndiaye et al. [109]	2023	ADEFGuard	Journal article	Dynamic	No	
Nguyen et al. [110]	2022	MANDO-GURU	Conference proceeding/workshop	Static	Yes	https://github.com/MANDO-Project/ge-sc-machine
Osei et al. [111]	2024	WIDENNET	Journal article	Static	No	
Pan et al. [112]	2021	ReDefender	Conference proceeding/workshop	Dynamic	No	
Pan et al. [113]	2023	Tx2TXT	Conference proceeding/workshop	Static	No	
Pasqua et al. [114]	2023	EtherSolve	Journal article	Static	Yes	https://github.com/SeUniVr/EtherSolve
Prasad and Ramachandram [115]	2022	N/A	Journal article	Dynamic	No	
Qian et al. [116]	2023	N/A	Conference proceeding/workshop	Static	Yes	https://github.com/Messi-Q/Cross-Modality-Bug-Detection
Qin et al. [117]	2023	N/A	Conference proceeding/workshop	Static	No	
Ren et al. [57]	2021	N/A (SCStudio)	Conference proceeding/workshop	Static	Yes	https://github.com/FISCO-BCOS/SCStudio
Ren et al. [58]	2021	SCStudio	Conference proceeding/workshop	Static	Yes	https://github.com/FISCO-BCOS/SCStudio
Rodler et al. [118]	2023	EF/CF	Conference proceeding/workshop	Dynamic	Yes	https://github.com/uni-due-syssec/efcf-framework
Rossini et al. [119]	2023	N/A	Conference proceeding/workshop	Static	No	
Sharma [120]	2023	N/A	Thesis or seminar paper	Both	No	
Shen and Li [121]	2023	N/A	Conference proceeding/workshop	Static	No	
Shou et al. [122]	2023	ItyFuzz	Conference proceeding/workshop	Dynamic	Yes	https://github.com/fuzzland/ityfuzz
Smaragdakis et al. [123]	2021	N/A	Conference proceeding/workshop	Static	No	
Sun et al. [124]	2023	SCOBERT	Conference proceeding/workshop	Static	No	
Sun et al. [125]	2023	ASSBert	Journal article	Static	No	

Tahir et al. [126]	2023	N/A	Conference proceeding/workshop	Static	No	
Tang et al. [127]	2021	N/A	Conference proceeding/workshop	Static	No	
Tong et al. [128]	2024	POSTER	Conference proceeding/workshop	Static	No	
Tran et al. [129]	2024	ChainSniper	Conference proceeding/workshop	Dynamic	No	
Wang et al. [130]	2023	N/A	Conference proceeding/workshop	Static	No	
Wang et al. [131]	2024	SliSE	Conference proceeding/workshop	Static	Yes	https://github.com/SliSE-SC/SliSE
Wang et al. [132]	2024	N/A	Conference proceeding/workshop	Static	No	
Wang et al. [133]	2024	SCVD-SA	Conference proceeding/workshop	Static	No	
Wang et al. [134]	2024	ReEP	Journal article	Static	Yes	https://github.com/ReEP-SC/ReEP
Wang et al. [135]	2024	DFier	Journal article	Dynamic	No	
Wu et al. [136]	2021	Peculiar	Conference proceeding/workshop	Static	Yes	https://github.com/wuhongjun15/Peculiar
Xu et al. [137]	2022	HAM-BiLSTM	Conference proceeding/workshop	Static	No	
Xu et al. [138]	2023	SoBERT-BiGRU-Attention	Conference proceeding/workshop	Static	No	
Xu et al. [139]	2024	ACBSC	Journal article	Static	No	
Xue et al. [140]	2021	Clairvoyance	Conference proceeding/workshop	Static	No	
Yan et al. [141]	2022	N/A	Conference proceeding/workshop	Static	No	
Yang et al. [142]	2023	NFTGuard	Conference proceeding/workshop	Static	Yes	https://github.com/NFTDefects/nftdefects
Yang et al. [143]	2024	BlockWatchdog	Conference proceeding/workshop	Dynamic	Yes	https://github.com/shuo-young/BlockWatchdog
Yang et al. [144]	2024	CrossFuzz	Journal article	Dynamic	Yes	https://github.com/yagol2020/CrossFuzz
Ye et al. [145]	2022	Vulpedia	Journal article	Static	No	
Ye et al. [146]	2024	FunFuzz	Journal article	Dynamic	Yes	https://github.com/MingxiYe/FunFuzz
Ye et al. [147]	2024	Midas	Conference proceeding/workshop	Dynamic	Yes	https://github.com/MingxiYe/Midas
Yin et al. [148]	2022	ConSym	Conference proceeding/workshop	Static	No	
Yu et al. [149]	2021	DeeSCVHunter	Conference proceeding/workshop	Static	Yes	https://github.com/MRdoulestar/DeeSCVHunter
Yu et al. [150]	2021	ReDetect	Conference proceeding/workshop	Static	No	
Yu et al. [151]	2023	PSCVFinder	Conference proceeding/workshop	Static	Yes	https://zenodo.org/records/8149025
Yuan et al. [152]	2023	N/A	Journal article	Static	Yes	https://github.com/davidyuan666/SmartContractDetection
Zeng et al. [153]	2023	SCVulBERT	Conference proceeding/workshop	Static	No	
Zhang et al. [154]	2022	MODNN	Journal article	Static	Yes	https://github.com/yangzuwjl/MODNN
Zhang et al. [155]	2023	SVScanner	Journal article	Static	No	
Zhang et al. [156]	2024	N/A	Journal article	Static	Yes	https://github.com/wobulijie10086/deection/tree/master
Zhou et al. [157]	2023	N/A	Journal article	Static	No	
Zhu et al. [158]	2023	GraBit	Conference proceeding/workshop	Static	No	
Zkik et al. [159]	2023	N/A	Conference proceeding/workshop	Dynamic	No	

Almakhour et al. [61] propose an approach to statically analyze composite Smart Contracts, i.e. contracts that depend on external contracts during execution. The contract to be analyzed is modeled as a finite state machine (FSM) and is then verified against specifications aimed at detecting multiple security issues, including reentrancy, denial of service and data exposure. Additionally, users may provide further security constraints using Computation Tree Logic (CTL) formulae.

Bose et al. [65] develop the tool *Sailfish*, which can detect reentrancy and transaction order dependence bugs. During analysis, the contract is first represented as a storage dependence graph (SDG), which models the control and data flow relations between the contract's storage variables as well as state-changing operations. Using graph queries, they determine whether there are any potential vulnerabilities in the contract. If so, a second analysis phase is started, which runs a so-called value-summary analysis, which helps initialize constraints for the following symbolic execution process and improves the tool's efficiency and scalability. The authors claim that the tool can detect delegate reentrancy as well as create-based reentrancy.

Braghin et al. [66] investigate the usage of Abstract State Machines (ASM) for Smart Contract analysis. They first model core elements of the Ethereum Virtual Machine, which are then applied when modeling specific Smart Contracts. They present their approach using a simplified version of the DAO contract containing a reentrancy vulnerability, showing that the reentrancy can be detected when appropriate rules and invariants are used. As of April 2024, the authors consider their research in this area to be an ongoing process.

Brent et al. [40] present *Vandal*, a security analysis tool that works on EVM bytecode. Vandal transforms the bytecode into logic relations, which then allows the security analysis to be expressed using the language Soufflé [160]. This means that users of the tool may extend Vandal by implementing further analyses using this language. Vandal provides a reentrancy analyzer which checks for `CALL` operations that forward any remaining gas and that can be reached recursively. To reduce false positives, Vandal checks whether `CALL` statements are protected using a reentrancy lock. The tool is available via GitHub [161].

Feist et al. [162] present a tool using a multi-stage procedure for vulnerability detection. *Slither* takes a contract's Abstract Syntax Tree (AST) as input and from that recovers additional information such as the Control Flow Graph (CFG). Then, the code is transformed into SlithIR, Slither's internal representation language. Different types of analyses are run on the resulting code, depending on which vulnerabilities should be detected. The first release of Slither was published in 2018 and the source code including updates is available via GitHub [163].

Garfatta et al. [84] extend their previous research in which Smart Contracts are represented as Colored Petri Nets (CPN), which allows their verification using Linear

Temporal Logic (LTL) formulae. They include modeling external calls and provide LTL formalizations of multiple vulnerabilities including reentrancy. Additionally, their approach allows users to add contract-specific properties to be verified using the contract's CPN model.

He et al. [91] present an approach similar to Garfatta et al.'s and propose modeling Smart Contracts as CPNs for reentrancy detection. Using a model of a simplified DAO contract as well as that of an attacker contract, they show that their proposed formal verification method can be used for vulnerability detection.

Luu et al. [35] introduce *Oyente* in 2016, a tool that is still present in current literature. Oyente is based on symbolic execution and uses path conditions to check contracts for reentrancy. For every `CALL` opcode, it is checked whether the path condition for the previous execution still holds if the variables are updated; if so, the contract is considered reentrant. While the repository was archived in May, 2023, the source code is still available via GitHub [164].

Mueller [36] presents his symbolic execution tool *Mythril* in 2018. Since then, the tool has been continuously updated and is available via GitHub [165]. Mythril consists of multiple modules, with each focusing on different vulnerability types. Reentrancy is part of the "External calls" module, which generates warnings when external calls that supply unrestricted gas are detected. Additionally, the "State Change External Calls" module warns users if any state changes following external calls are detected.

Pasqua et al. [114] attempt to reconstruct precise Control Flow Graphs from compiled Smart Contracts and recommend their approach be used in further analysis tools. To validate their claims, they implement a tool called *EtherSolve*, which detects reentrancy by searching for locations in the graph where `SSTORE` opcodes are reachable from unsafe `CALL` blocks. The authors state that the tool performs slightly worse compared to *Slither*, but better than other similar tools used in their evaluation.

Schneidewind et al. [166] present *eThor*, a sound and automated static analyzer for EVM bytecode published in 2020. This means that eThor can guarantee the absence of a vulnerability, thus resulting in no false negatives. The tool uses a reachability analysis which is realized by Horn clause resolution, which allows the verification of various security properties. To prove a contract is free from any reentrancy bugs, eThor thus tries to verify the single-entrancy property. The authors experimentally compare eThor to the analysis tool ZEUS [167], which reveals that eThor outperforms the tool. The eThor source code is available online [168].

Smaragdakis et al. [123] introduce "symvalic analysis", which is a type of value-flow analysis that allows symbolic expressions. While symvalic analysis is neither sound nor complete, the authors intend to cover more behaviors than symbolic evaluation

would, while keeping the unrealizable behavior predictions at a minimum. Symvalic analysis produces better precision and recall results for reentrancy bugs than the tool Mythril, with the drawback that the symvalic analysis tool supports a smaller amount of vulnerabilities.

Tang et al. [169] recommend tool developers to distinguish between benign and malicious ("offensive") reentrancy. They propose a Datalog-based approach for reentrancy detection, which they apply on more than 459 million transactions. They compare their results with other tools and find that their Datalog-based approach produces a significantly lower number of false positives.

Wang et al. [130] propose a symbolic execution vulnerability detection tool. They hope to improve the tool efficiency by pruning the contracts' Control Flow Graphs, meaning that less paths need to traversed during the symbolic execution. Only paths containing "critical instructions" such as `CALL` are considered for analysis. Compared to other symbolic execution tools, this approach manages to reduce the execution times while also improving the detection rates for multiple types of vulnerabilities.

Wang et al. [131] introduce a tool called *SliSE*, which divides the detection process into to steps. Firstly, during the "Warning Search" stage, program slicing is used to extract critical paths, while in the following "Symbolic Execution Verification" stage, symbolic execution is performed to verify the reachability of the critical paths, which then determines whether the contract contains a reentrancy vulnerability or not. In their evaluation, the authors conclude that their approach is well-suited for reentrancy detection, as SliSE achieves a recall rate of over 90% for reentrancy on the selected test data sets.

Wang et al. [134] design a reentrancy detection tool called *ReEP* which is specifically aimed at reducing the number of false positives. Again, the tool consists of a CFG pruning phase and a symbolic execution phase. In addition to that, ReEP allows users to integrate other Smart Contract vulnerability analysis tools into ReEP. By merging the results of all tools, the results' precision is greatly improved.

Xue et al. [140] propose a tool called *Clairvoyance*, which allows cross-function and cross-contract analysis to detect reentrancy vulnerabilities. A static taint analysis is performed on a cross-contract interprocedural Control Flow Graph (XCFG) to determine paths that might contain a vulnerability. So-called Path Protection Techniques are then considered during a symbolic analysis phase to remove infeasible paths. Using this approach, the authors report that Clairvoyance outperforms three other static analysis tools in terms of precision and recall.

Yang et al. [142] build a tool specifically meant to analyze Smart Contracts used as non-fungible tokens (NFTs). *NFTGuard* operates on bytecode and attempts to recover

source code-level features in order to improve the detection quality. The analysis phase uses symbolic execution to detect a total of five vulnerabilities, one of them being reentrancy. The authors report that NFTGuard achieves an overall precision of 92.6%.

Ye et al. [145] base their tool's vulnerability detection process on so-called vulnerability signatures. *Vulpedia*'s vulnerable signatures describe code patterns that point to a vulnerability being present, while benign signatures represent safe coding patterns. The signatures are abstracted by running existing vulnerability detection tools on a data set containing Solidity Smart Contracts. The contracts falsely reported to contain a vulnerability are used for benign signature abstraction, while the true positives form the basis of the vulnerable signature abstraction. The resulting signatures are then matched against unseen contracts. Only contracts that match a vulnerable signature but no benign signature are considered vulnerable.

Yin et al. [148] propose a mitigation method for symbolic execution-based detectors, which is aimed at reducing the number of false positives. *ConSym* introduces multiple constraints that are experimentally evaluated by integrating them into the tool *Osiris* [170]. The results show that ConSym reduces the number of false positives while retaining the number of true positives.

Yu et al. [150] introduce an analysis tool that operates on bytecode level, called *ReDetect*. The tool builds a CFG which is analyzed using symbolic execution. To reduce the number of false positives, Yu et al. introduce five path filters that constrain the symbolic execution. Using this approach, ReDetect achieves better accuracy than two other reentrancy detection tools.

CHAPTER 6

Experiment Design

As mentioned previously, we aim to answer our final research question, *"What are the discrepancies between the analysis results of the open-source tools and what are their causes?"*, by conducting an experimental study. In this experiment, selected open-source analysis tools are run on a predefined data set which is specifically created to cover the different types of reentrancy and its countermeasures as discussed in chapter 3. This chapter describes the general setup of the experiment as well as the structure and content of the test data, while the following chapter presents the experiment results including an evaluation.

6.1 Tool selection

As briefly mentioned in the previous chapter, the tools Oyente [35], Mythril [36], Slither [162], and Vandal [40] often serve as baseline when evaluating new tools or approaches. For this reason, we select two of the tools, Oyente and Mythril, to be part of the experimental study. As all four of the identified reentrancy types are to be analyzed, we additionally add the newer tool Sailfish [65], which is claimed to recognize both delegate reentrancy as well as create-based reentrancy. To add an approach that differs from the three tools selected so far, eThor [166] is the fourth and last tool to be examined in the experiment. The following paragraphs give a short recap of the selected tools' approaches.

Oyente is a symbolic execution tool that uses path conditions to check for reentrancy and focuses on `CALL` opcodes. The experiment uses an adapted version of Oyente that is based on version 0.2.7.

Mythril is a symbolic execution tool that contains two modules relevant for reentrancy, "External Calls" and "State Change External Calls". Warnings are generated for exter-

nal calls with unrestricted gas, as well as state changes following external calls. The experiment uses the version 0.23.15 of Mythril.

Sailfish uses a hybrid approach containing two phases, where first, graph queries are used to reduce the number of instructions that need to be analyzed in the second phase, which consists of symbolic evaluation in combination with a so-called value-summary analysis. The experiment uses an adapted version of the code published on GitHub.

eThor is a sound static analyzer which attempts to prove that a given contract is reentrancy-free, and as such does not produce false negatives. The experiment uses the 2023 version of the tool.

6.2 Experiment setup

Based on the information gained in the previous research, the decision was made to evaluate four Smart Contract analysis tools regarding their behavior depending on the different types of reentrant and non-reentrant contracts. Due to continuously changing language features and the introduction of new opcodes which might impact analysis results, the tools are tested on multiple versions of the Solidity language, namely versions 0.4 to 0.8.

As, to the best of our knowledge, no publicly available data set covering the different types of reentrancy existed at the time of writing, we created such a data set which acts as a ground truth in the experiment and is further described in the following subsection. To allow simpler reproducibility of the experiment, the open-source framework *SmartBugs 2.0.8*[1] was used, which provides a uniform interface for executing the selected tools. SmartBugs 2.0.8 allows out-of-the-box execution of eThor, Mythril and Oyente; to allow analyzing Smart Contracts using Sailfish, the tool first had to be manually integrated. This included an update of the tool's Docker image, as it was decided to use *solc-select* for switching between Solidity versions instead of the tool's original process, which only allowed the usage of a limited number of versions. The updated Docker image is publicly available and can be found at `https://hub.docker.com/r/lmfue/sailfish`. Additionally, an improved version of Oyente[2], which allows analyzing Solidity version 8 bytecode files, was provided by the maintainers of SmartBugs and is used in the experiment. All tools were executed using SmartBugs' default arguments.

SmartBugs saves the analysis results in the form of the tools' raw output and provides additional information in a separate JSON file, which includes data such as the duration of the analysis and the exact command the tool was executed with. These files were parsed and necessary data were extracted in order to generate standardized CSV files, allowing comparability of the results. The final results are provided in chapter 7.

[1] `https://github.com/smartbugs/smartbugs/releases/tag/v2.0.8`
[2] `https://github.com/smartbugs/oyente/tree/update`

An overview of the tools' capabilities regarding source code and bytecode analysis is given in figure 6.1 and figure 6.2.

Source code	**v0.4**	**v0.5**	**v0.6**	**v0.7**	**v0.8**
eThor	×	×	×	×	×
Mythril	✓	✓	✓	✓	✓
Oyente	✓	✓	✓	✓	×
Sailfish	✓	✓	✓	×	×

Table 6.1: Overview of the tools' source code analysis capabilities

Bytecode	**v0.4**	**v0.5**	**v0.6**	**v0.7**	**v0.8**
eThor	✓	✓	✓	✓	✓
Mythril	✓	✓	✓	✓	✓
Oyente	✓	✓	✓	✓	✓
Sailfish	×	×	×	×	×

Table 6.2: Overview of the tools' bytecode analysis capabilities

6.3 Test data and ground truth

To allow a comprehensive comparison of the different types of reentrancy, a data set comprised of a total of 260 Solidity Smart Contracts was written. Compiling these source code files results in a total of 390 additional bytecode files that are also used in the experiment. The distribution of files based on the different reentrancy types is displayed in table 6.3. Many of the contracts in the data set are based on example contracts published by Rodler et al. on GitHub[3]. To ensure a more uniform data set, each Solidity version used is represented by the same minor version; that is, the data set uses Solidity 0.4.26, 0.5.17, 0.6.12, 0.7.6 and 0.8.0. Between versions, the corresponding test data files only differ in the cases where certain functions were deprecated or adapted; for example, where version 0.5 uses `msg.sender.call.value(500000)("")`, version 0.6 uses the equivalent `msg.sender.call{value: 500000}("")`.

The data set is available at `https://github.com/lmfuerst/DA_Testdaten`.

	# Solidity files	**# bytecode files**	**Total**
Create-based reentrancy	65	130	195
Cross-function reentrancy	65	65	130
Delegated reentrancy	65	130	195
Same-function reentrancy	65	65	130

Table 6.3: Number of files in the data set.

[3] `https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns`

As can be seen, the number of files per reentrancy type varies, which is explained by their respective properties. Both create-based and delegate-based reentrancy require two contracts to interact with each other, which can be included in the same Solidity file, yet require separate bytecode files.

Each file is mapped to a truth value which describes whether the contract is considered to be vulnerable or not. Thus, contracts which are reentrant yet non-vulnerable are considered secure. Table 6.4 gives an overview of the final test data classification, irrespective of reentrancy type.

		Insecure?
No	Non-reentrant	False
	Reentrant code parts	False
Possibly	Send	False
	Transfer	False
Reentrant	Vulnerable w/ Ether loss	True
	Vulnerable w/o Ether loss	True
	Non-vulnerable	False

Table 6.4: Classification of the test data.

Note that for delegated and create-based reentrancy, i.e. the types where two separate bytecode files need to be analyzed, the corresponding results were combined. This means that the final analysis result is considered to be "insecure" only if both contracts are detected as being vulnerable.

6.4 Metrics

As we deal with a binary classification problem, the decision was made to use the metrics *accuracy*, *precision*, *recall*, *specificity* and *error rate*. These metrics, except for the error rate, can be directly calculated when dividing the analysis results into a confusion matrix such as shown in table 6.5.

		Predicted	
		Insecure	**Secure**
Actual	**Insecure**	True Positive (TP)	False Negative (FN)
	Secure	False Positive (FP)	True Negative (TN)

Table 6.5: Confusion matrix used to calculate evaluation metrics.

True positives and true negatives are contracts which are correctly identified to be vulnerable ("positive") or non-vulnerable ("negative"), respectively. On the other hand, non-vulnerable contracts which are detected to be insecure are considered false positives, while vulnerable contracts which are reported to be non-vulnerable are called false negatives. Should an analysis tool not produce valid output, for example if an exception

occurs, then the result is considered an error and the contract is not included in the confusion matrix. The error rate, i.e. the fraction of contracts that could not be classified by a tool, reflects such outcomes.

Accuracy is calculated using the formula

$$\frac{TP + TN}{TP + TN + FP + FN}, [171]$$

thus describing the fraction of contracts that were correctly recognized as either secure or insecure and allowing a high-level comparison of the tools' performances.

The precision score can give insight into whether a tool tends to be overly cautious during the classification process, as the formula

$$\frac{TP}{TP + FP} [171]$$

describes how many of the contracts deemed insecure indeed contained a vulnerability.

Recall,

$$\frac{TP}{TP + FN}, [171]$$

describes the fraction of vulnerable contracts that were correctly classified by the tool. This metric is also known as true positive rate or sensitivity.

Similarly, specificity,

$$\frac{TN}{TN + FP}, [171]$$

describes the fraction of secure contracts that were correctly classified by the tool.

Note that the metrics are sensitive to the balancing of the data set and each highlight specific strengths and weaknesses of the classifiers, which needs to be taken into account in this case, as the majority of contracts the tools are tested on are considered secure (cf. table 6.4). Thus, to evaluate the performance of the tools, we will examine and compare all presented metrics to allow a more accurate and comprehensive analysis, instead of focusing on a single metric.

CHAPTER 7

Evaluation

This chapter provides an evaluation of the experiment results with the aim of identifying differences between the tools, including strengths and weaknesses. Where possible, the causes of such discrepancies are investigated. To allow a closer look at the properties of the tested tools, the results are split into multiple groups, which are evaluated separately. Both the type of analysis (source code or bytecode analysis) as well as the contracts' Solidity versions are taken into account. Additionally, the tools' performance depending on the contracts' type of reentrancy is displayed. All experiment results used in this chapter are available at `https://github.com/lmfuerst/da-results-to-csv`. The metrics used in this evaluation are described in the previous chapter and consist of accuracy (in- and excluding invalid analysis outputs), precision, recall, specificity and error rate.

Note that, as described in the previous chapter, not all tools support both source code and bytecode analysis for all Solidity versions.

7.1 Results by file type

To gain information on whether the two tools that are capable of analyzing both source code and bytecode files perform better with either of the types, the results are examined separately. A combination of the two file types, i.e. an overview of all results, is presented as well.

7.1.1 Source code analysis

Three of the tools, Mythril, Oyente and Sailfish, were used to analyze the Solidity source code files. A total of 576 analyses were run, with 240 of them falling to Mythril, 192 to Oyente, and 144 to Sailfish. The reason for the varying analysis numbers is the lack of version support from both Oyente and Sailfish. Table 7.1 shows that Oyente achieved the

highest overall accuracy while maintaining a low error rate. While Sailfish also manages to provide decent accuracy, it suffers from a high error rate, as 15% of the source code analysis results had to be discarded. The highest recall achieved is 39%, meaning that none of the tools manage to consistently detect vulnerable contracts, but instead have a high number of false negatives. As highlighted in the table, Oyente consistently provides good results when compared to the other tools, but comes at the cost of not accepting Solidity version 0.8 source code for analysis.

	Accuracy w/o error	Accuracy w/ error	Precision	Recall	Specificity	Error rate
eThor	-	-	-	-	-	-
Mythril	0.68	0.66	0.50	0.38	0.83	0.03
Oyente	0.76	0.75	0.75	0.33	0.95	0.02
Sailfish	0.75	0.63	0.60	0.39	0.89	0.15
Overall	***0.72***	***0.68***	***0.58***	***0.36***	***0.88***	***0.06***

Table 7.1: Experiment results for source code analysis

7.1.2 Bytecode analysis

Bytecode analysis results are available for eThor, Mythril, and Oyente, which are all capable of analyzing the Solidity versions 0.4 to 0.8. As shown in table 7.2, the tool eThor makes true on its claim that it can guarantee the absence of reentrancy in a contract, and thus results in zero false negatives and a 100% recall rate. Conversely, it suffers from low specificity, meaning that it produces a high number of false positives. Oyente outperforms Mythril when disregarding invalid output, however an 11% error rate, as compared to Mythril's 0% error rate, means that Mythril's overall accuracy is higher.

	Accuracy w/o error	Accuracy w/ error	Precision	Recall	Specificity	Error rate
eThor	0.44	0.31	0.33	1.00	0.23	0.30
Mythril	0.70	0.70	0.52	0.45	0.82	0.00
Oyente	0.76	0.67	0.74	0.38	0.94	0.11
Sailfish	-	-	-	-	-	-
Overall	*0.65*	*0.56*	*0.44*	*0.56*	*0.69*	*0.14*

Table 7.2: Experiment results for bytecode analysis

7.1.3 Overall results

Figure 7.1 highlights the fact that there are significant differences between the accuracies of the results produced by the four tools. While Mythril, Oyente and Sailfish are relatively similar in terms of overall accuracy, their error rates range from just 2% to 15%. The overall results confirm that eThor is suitable for specific cases where individual contracts need to be proven reentrancy-free, but other tools will generally be better suited for most users due to their higher versatility and improved accuracy.

Table 7.3 presents an overview of the combined analysis results, including both source code and bytecode analysis. This impacts Mythril and Oyente, as both eThor and Sailfish

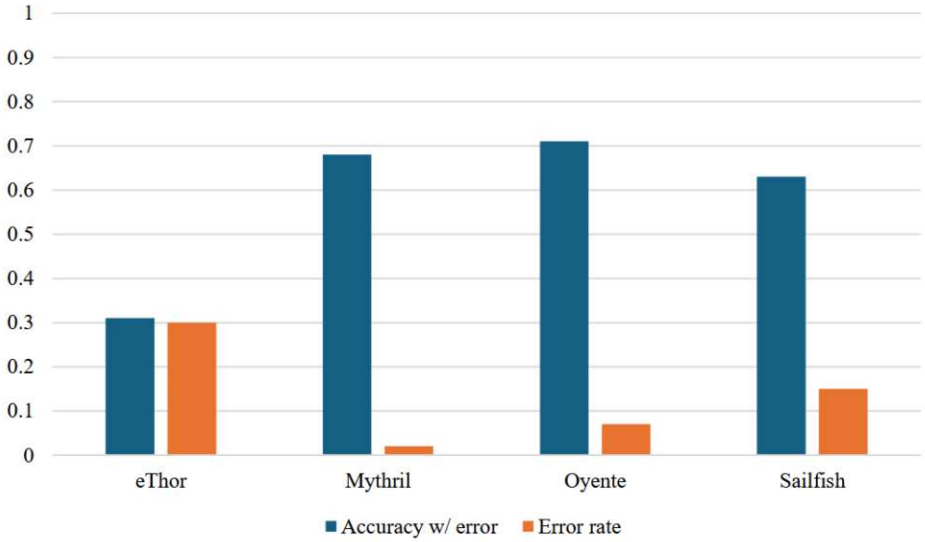

Figure 7.1: Comparison of overall accuracy and error rate

only support one type of analysis. Considering all data, Oyente achieves the highest overall accuracy, precision and specificity, meaning that it does not tend to misclassify secure contracts as vulnerable. However, with 36%, Oyente has the lowest recall rate out of the four tools, meaning that it produces a high number of false negatives, i.e. it tends to consider reentrant contracts as non-vulnerable. This is in contrast to eThor's results, which aims to not produce any false negatives. Thus, a large number of contracts are considered potentially unsafe, leading to a recall of 100% while struggling with a specificity of just 23%.

	Accuracy w/o error	Accuracy w/ error	Precision	Recall	Specificity	Error rate
eThor	0.44	0.31	0.33	1.00	0.23	0.30
Mythril	0.69	0.68	0.51	0.41	0.82	0.02
Oyente	0.76	0.71	0.74	0.36	0.94	0.07
Sailfish	0.75	0.63	0.60	0.39	0.89	0.15
Overall	*0.69*	*0.62*	*0.49*	*0.47*	*0.78*	*0.10*

Table 7.3: Experiment results for source code and bytecode analysis

7.2 Results by reentrancy type

Figure 7.2 shows each tool's results split by the reentrancy types identified in chapter 3. Again, we can see that eThor consistently performs worse than the other tools when comparing the accuracy values, while Oyente and Mythril generally achieve comparatively high accuracies, regardless of reentrancy type. More detailed results are provided in the following subsections.

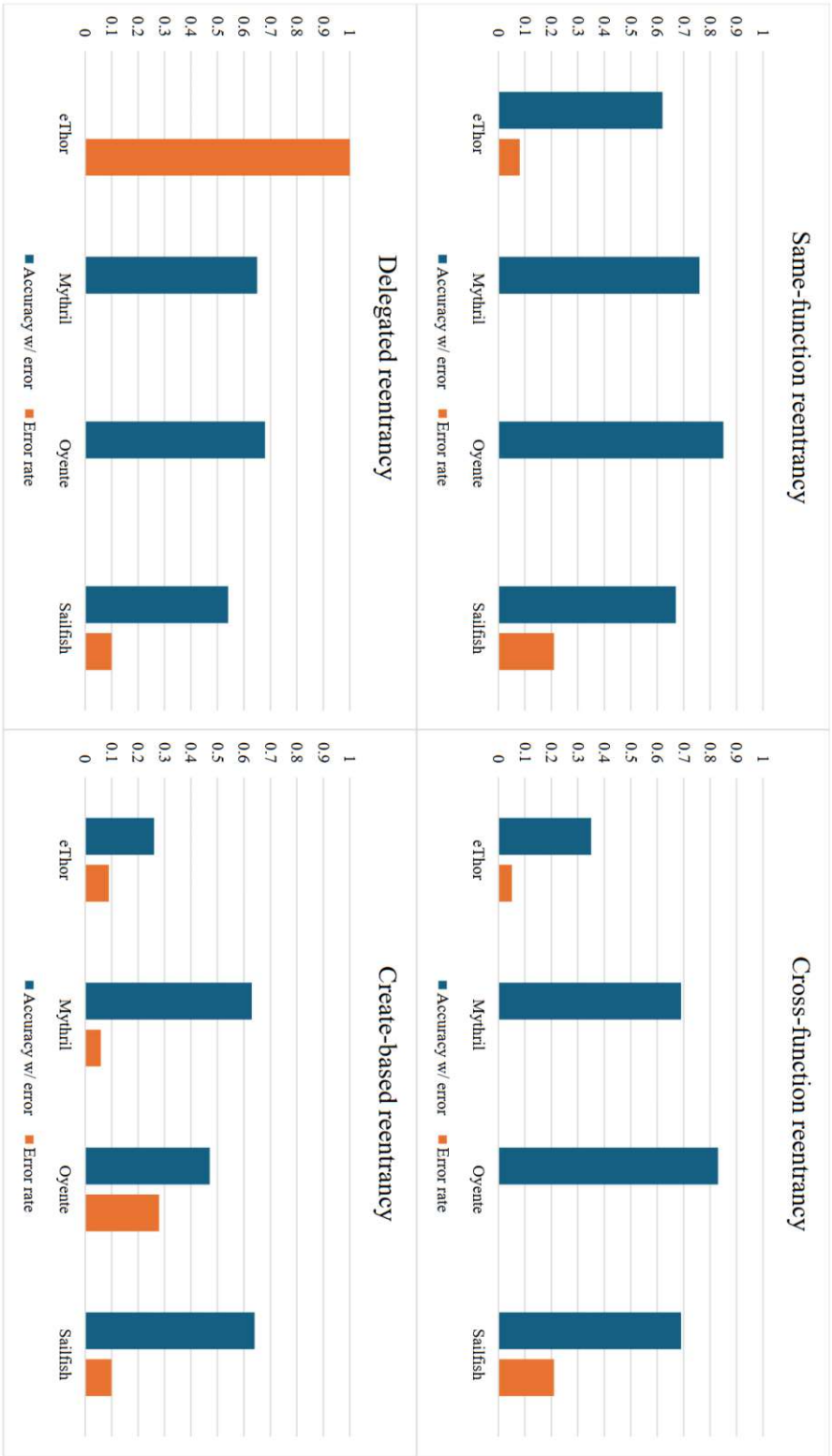

Figure 7.2: Comparison of accuracy and error rate by reentrancy type

7.2.1 Same-function reentrancy

Same-function reentrancy can be considered the "standard" form of the vulnerability, which may be an explanation as to why the average accuracy of the tools' results for this type exceeds the corresponding value of the other reentrancy types. Oyente provides the best results for all metrics except recall, where it only offers a rate of 50%, meaning that half of the vulnerable contracts in the data set were not detected. In comparison, Mythril achieves a 90% recall rate, but tends to misclassify non-reentrant contracts as vulnerable, bringing its precision score down to 57%. Compared to the other types, eThor performs best on same-function reentrancy contracts, where its accuracy score is at 62% while having an error rate of only 8%. Even though Sailfish has a score of 100% for both precision and specificity, it suffers from low recall, thus lowering its accuracy to 67%.

	Accuracy w/o error	Accuracy w/ error	Precision	Recall	Specificity	Error rate
eThor	0.67	0.62	0.43	1.00	0.56	0.08
Mythril	0.76	0.76	0.57	0.90	0.70	0.00
Oyente	0.85	0.85	1.00	0.50	1.00	0.00
Sailfish	0.84	0.67	1.00	0.38	1.00	0.21
Overall	*0.78*	*0.75*	*0.61*	*0.73*	*0.80*	*0.04*

Table 7.4: Experiment results for same-function reentrancy

7.2.2 Cross-function reentrancy

Cross-function reentrancy requires at least two functions to be called to trigger the vulnerability. According to our results, this generally makes it harder for the tools to detect reentrant contracts, as three out of four tools achieve a lower accuracy compared to same-function reentrancy. The only exception is Sailfish, which improves its accuracy by 2% and the recall by 12% while keeping the same error rate. Interestingly, Mythril does not mark any of the cross-function reentrancy contracts as vulnerable, leading to undefined precision, a recall of 0%, and a specificity of 100%. This example also shows the necessity of taking into account metrics other than accuracy, as due to the high number of true negative contracts in the data set, Mythril still achieves an accuracy of 69%. Besides Mythril, Oyente and Sailfish also do not produce any false positives in their analyses, thus setting their specificity scores to 100% as well. However, they correctly detect some of the vulnerable contracts, resulting in recall values of 44% and 50%, respectively. The tool eThor again reaches perfect recall while struggling with accuracy, precision and specificity.

	Accuracy w/o error	Accuracy w/ error	Precision	Recall	Specificity	Error rate
eThor	0.37	0.35	0.32	1.00	0.11	0.05
Mythril	0.69	0.69	undefined	0.00	1.00	0.00
Oyente	0.83	0.83	1.00	0.44	1.00	0.00
Sailfish	0.87	0.69	1.00	0.50	1.00	0.21
Overall	*0.70*	*0.68*	*0.49*	*0.37*	*0.84*	*0.03*

Table 7.5: Experiment results for cross-function reentrancy

7.2.3 Delegated reentrancy

As the DELEGATECALL opcode is not supported by eThor, it cannot analyze contracts containing a delegated reentrancy, and thus produces a 100% error rate for this reentrancy type.

Both Mythril and Oyente manage to keep their error rates at 0%, but still do not perform very well when considering the other metrics. Oyente outdoes Mythril when it comes to accuracy, precision and specificity, but has a considerably lower recall rate than Mythril, indicating that it misclassifies a significant portion of the vulnerable contracts. As the tools' highest precision value for this reentrancy type is at 47%, we can additionally conclude that there is a large number of false positives, meaning that the tools have a tendency to mark contracts containing a delegatecall as vulnerable even if they are not. Disregarding eThor, Sailfish achieves the worst results, with an accuracy as low as 54%. A precision of 25% as well as a recall of just 20% further confirm that Sailfish is not well suited to detect delegated reentrancy. Only its specificity of 76% can be considered a reasonably good result for this reentrancy type.

	Accuracy w/o error	Accuracy w/ error	Precision	Recall	Specificity	Error rate
eThor	undefined	0.00	undefined	undefined	undefined	1.00
Mythril	0.65	0.65	0.45	0.75	0.60	0.00
Oyente	0.68	0.68	0.47	0.42	0.79	0.00
Sailfish	0.60	0.54	0.25	0.20	0.76	0.10
Overall	*0.65*	*0.52*	*0.44*	*0.55*	*0.70*	*0.20*

Table 7.6: Experiment results for delegated reentrancy

7.2.4 Create-based reentrancy

Like delegated reentrancy, create-based reentrancy requires two contracts; a contract that initiates the creation, and the contract to be created. Again, this seems to pose a problem to the tools, as the highest accuracy is achieved by Sailfish and is at 64% for this reentrancy type. None of the test contracts are considered secure by eThor in this case, leading to a specificity of 0%. Conversely, Mythril and Oyente both achieve a specificity of 100% with a recall of 0%, as they consider all contracts to be vulnerability-free. Sailfish finds a better balance and provides precision, recall and specificity scores of at least 50%. Its error rate of 10% is slightly worse than that of eThor and Mythril, and negatively impacts its overall accuracy.

	Accuracy w/o error	Accuracy w/ error	Precision	Recall	Specificity	Error rate
eThor	0.29	0.26	0.29	1.00	0.00	0.09
Mythril	0.67	0.63	undefined	0.00	1.00	0.06
Oyente	0.65	0.47	undefined	0.00	1.00	0.28
Sailfish	0.71	0.64	0.60	0.50	0.83	0.10
Overall	*0.60*	*0.51*	*0.33*	*0.23*	*0.77*	*0.15*

Table 7.7: Experiment results for create-based reentrancy

7.3 Results by Solidity version

As Solidity is constantly under development, version updates may have introduced changes to its syntax and semantics that impact the results of Smart Contract analysis tools. Therefore, we split the results by Solidity version, 0.4 to 0.8, and investigate them separately.

Figure 7.3 presents a side-by-side overview of each tool's results per version, again showing the overall accuracy as well as the error rate. The graphs show that eThor consistently has a relatively high error rate, which jumps to nearly 50% at version 0.8. Conversely, Mythril's error rate per version remains consistently below 5%, dropping to 0% at 0.8. Oyente's accuracy shows a downward trend, while also increasing its error rate with each version. Sailfish only allows the versions 0.4 to 0.6, but suffers a sharp drop in accuracy as well as an increase in error rate with version 0.6. Based on the metrics presented in the figure, Mythril and Oyente seem to be the best suited tools for reentrancy detection across all five versions.

7.3.1 Solidity version 0.4

Considering all metrics, Oyente, Mythril and Sailfish achieve very similar results for version 0.4. Oyente, with an accuracy of 77%, narrowly surpasses the other two tools by 4% and 2%, respectively. Recall is the only metric where the three tools do not perform well, with results ranging from 38% to 44%. Again, eThor reaches a perfect recall score of 100%, but remains below 50% for all other metrics. Its error rate of 29% also negatively impacts its overall accuracy, which is at 31%, i.e. more than 40 percentage points lower than that of the other tools.

	Accuracy w/o error	Accuracy w/ error	Precision	Recall	Specificity	Error rate
eThor	0.43	0.31	0.32	1.00	0.22	0.29
Mythril	0.75	0.73	0.67	0.38	0.91	0.02
Oyente	0.77	0.77	0.75	0.38	0.94	0.00
Sailfish	0.75	0.75	0.64	0.44	0.89	0.00
Overall	*0.72*	*0.68*	*0.54*	*0.46*	*0.83*	*0.05*

Table 7.8: Experiment results for Solidity version 0.4

7.3.2 Solidity version 0.5

The tools' results for version 0.5 are slightly lower than those of 0.4, with the exception of eThor, which improves its accuracy to 33% while lowering the error rate to 27%. While Oyente still achieves the best scores in regards to accuracy, precision and specificity, it also increases its error rate to 5%, making Mythril the best tool in regards to error rate, having just 2 percentage points. Mythril also manages to raise its recall to 50%, but at the same time has considerably lower scores for precision and specificity in comparison to version 0.4. Sailfish produces a higher number of invalid outputs, which negatively impacts all metrics, if only slightly. Again, eThor struggles with specificity and precision

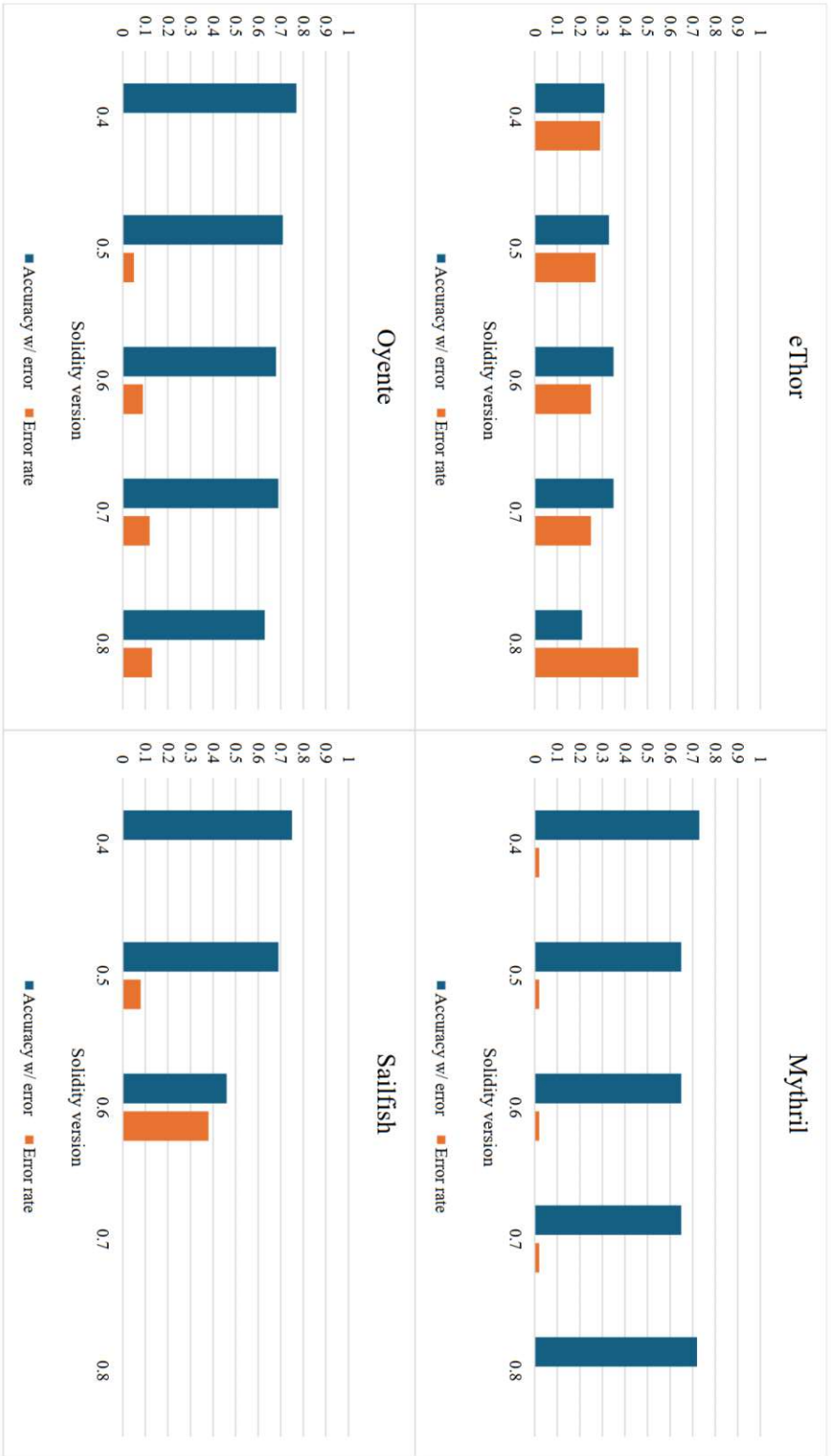

Figure 7.3: Comparison of accuracy and error rate by Solidity version

while reaching a perfect recall, while the other tools do not achieve a recall higher than 50%.

	Accuracy w/o error	Accuracy w/ error	Precision	Recall	Specificity	Error rate
eThor	0.45	0.33	0.34	1.00	0.22	0.27
Mythril	0.68	0.65	0.47	0.50	0.74	0.02
Oyente	0.75	0.71	0.71	0.32	0.94	0.05
Sailfish	0.75	0.69	0.60	0.43	0.88	0.08
Overall	*0.68*	*0.63*	*0.48*	*0.49*	*0.76*	*0.08*

Table 7.9: Experiment results for Solidity version 0.5

7.3.3 Solidity version 0.6

Mythril's error rate remains unchanged, while eThor again improves both its accuracy and error rate by 2%. Oyente and Sailfish both experience upward jumps in error rate, with Sailfish even producing invalid output for more than a third of the test contracts, thus dropping its accuracy to under 50%. Although Oyente and Mythril have relatively similar accuracies, Mythril achieves considerably lower precision and specificity scores, while at the same time surpassing Oyente in terms of recall.

	Accuracy w/o error	Accuracy w/ error	Precision	Recall	Specificity	Error rate
eThor	0.46	0.35	0.36	1.00	0.22	0.25
Mythril	0.67	0.65	0.47	0.50	0.74	0.02
Oyente	0.75	0.68	0.73	0.35	0.94	0.09
Sailfish	0.75	0.46	0.50	0.25	0.92	0.38
Overall	*0.68*	*0.58*	*0.48*	*0.49*	*0.76*	*0.14*

Table 7.10: Experiment results for Solidity version 0.6

7.3.4 Solidity version 0.7

As Sailfish does not support version 0.7, no results for this tool are available. Oyente manages to improve its results compared to the previous version, despite its error rate increasing to 12%. Its high specificity and low recall again confirm that the tool has a tendency to over-classify contracts as secure. Despite being close in accuracy to Oyente, Mythril again struggles with precision and does not improve its score compared to version 0.6. It's however still in the lead when it comes to error rate, reaching a near-perfect 2% score. The metrics for eThor do not change with this version update; its recall remains at 100%, while accuracy, precision and specificity are below 40%.

7.3.5 Solidity version 0.8

Version 0.8 is the first version where Mythril achieves better overall accuracy than Oyente, improving it to 72%, and only producing valid output. In comparison, Oyente's error rate goes up to 13%, while its accuracy drops to 63%. Oyente does outperform Mythril in regards to recall, however both tools achieve very low results, having rates of 29% and

	Accuracy w/o error	Accuracy w/ error	Precision	Recall	Specificity	Error rate
eThor	0.46	0.35	0.36	1.00	0.22	0.25
Mythril	0.67	0.65	0.47	0.50	0.74	0.02
Oyente	0.78	0.69	0.80	0.41	0.95	0.12
Sailfish	-	-	-	-	-	-
Overall	*0.68*	*0.61*	*0.49*	*0.55*	*0.74*	*0.10*

Table 7.11: Experiment results for Solidity version 0.7

19%, respectively. This means that despite the comparatively high accuracies, Mythril and Oyente both misclassify a large number of reentrant contracts. The tool eThor struggles with producing valid results for version 0.8 contracts and increases its error rate to 46%. This, in turn, also affects accuracy and precision, dropping them by more than ten percentage points. However, eThor still keeps its 100% recall score by not producing any false negatives.

	Accuracy w/o error	Accuracy w/ error	Precision	Recall	Specificity	Error rate
eThor	0.39	0.21	0.23	1.00	0.26	0.46
Mythril	0.72	0.72	0.67	0.19	0.96	0.00
Oyente	0.73	0.63	0.67	0.29	0.94	0.13
Sailfish	-	-	-	-	-	-
Overall	*0.67*	*0.57*	*0.41*	*0.29*	*0.83*	*0.15*

Table 7.12: Experiment results for Solidity version 0.8

7.4 Summary and interpretation

7.4.1 Reentrancy type detection

Based on the results presented in this chapter, we evaluate whether the tools are suited for detecting all four of the reentrancy types discussed in this work. We consider a tool suitable for detection if it manages to correctly flag at least half of the reentrant contracts, while at the same time keeping the number of misclassified secure contracts low. Therefore, we establish the following criteria:

- a recall of at least 50%,
- and a precision of at least 40%.

Combining both metrics is necessary as 100% recall can be reached by simply classifying all test contracts as vulnerable, which however does not bring any value to users, as such a result would require all contracts to be checked manually. Additionally setting a minimum precision forces the tool to attempt to discern between vulnerable and secure contracts, as precision is negatively impacted by false positives, which there would be a notable amount of if the tool were to classify every file as vulnerable.

Table 7.13 shows that despite the fact that Oyente and Mythril reached the best accuracy scores for most reentrancy types, this does not mean that they are actually well-suited to detect them all.

	Same-function	**Cross-function**	**Delegated**	**Create-based**
eThor	✓	×	×	×
Mythril	✓	×	✓	×
Oyente	✓	×	×	×
Sailfish	×	✓	×	✓

Table 7.13: Overview of reentrancy type detection by tool

Interestingly, Oyente only fulfills the criteria for same-function reentrancy. This is due to the fact that Oyente consistently achieves low recall values, with the value for same-function reentrancy being at only 50%. Low recall values are caused by a large number of false negatives, i.e. Oyente struggles with recognizing reentrant contracts. Its high accuracy is explained by the fact that the majority of test contracts do not contain a reentrancy vulnerability, and as such is not heavily impacted by false negatives.

Mythril, on the other hand, suffers from lower precision for same-function reentrancy and delegated reentrancy, but still manages to fulfill the criteria for the two types. For both cross-function reentrancy and create-based reentrancy, however, Mythril does not classify any of the test contracts as vulnerable, resulting in undefined precision and a recall of 0%.

The precision achieved by eThor only suffices in the case of same-function reentrancy; for all other types, eThor produces either too many false positives or too many errors. Despite the fact that eThor performs poorly when directly compared to the other tools, the results do prove that the tool can guarantee absence of the vulnerability for all contracts identified as vulnerability-free, making it interesting for certain niche problems.

Sailfish, surprisingly, does not manage to fulfill the criteria for same-function reentrancy, caused by a recall of only 38%, despite this type being the best-known form of reentrancy. Still, Sailfish is the only tool considered suitable for cross-function and create-based reentrancy, although in both cases the recall is limited to 50%. The authors' claim that Sailfish is suitable for the detection of delegated reentrancy is not supported by our results.

7.4.2 Version influence

The first Solidity version which was used in the test contracts, 0.4.26, was released in April 2016, while version 0.8.0 was released in December 2020 [16]. Our results show that all tools but Mythril are negatively impacted by handling increasing version numbers. Therefore, this subsection collects some of the larger language changes that could be relevant to the negative impact on the analysis results.

Solidity version 0.5 changes

Solidity version 0.5 introduces syntactic and semantic changes to `delegatecall()` and `call()`, which now return additional data and are limited in terms of accepted arguments [172]. Furthermore, users now explicitly have to declare addresses as `payable` should they wish to transfer ether to them.

Solidity version 0.6 changes

Version 0.6 splits up the previously unnamed fallback function into a function defined using the keyword `fallback`, called when no other function in the contract matches, and `receive`, called whenever the call data is empty and only if the function is present [173]. It also introduces an update that concerns the `CREATE2` opcode.

Solidity version 0.7 changes

The syntax of external function calls and contract creation is updated in version 0.7, and Solidity now allows Unicode characters [174]. Version 0.7 also introduces file-level constants and functions, i.e. function definitions outside of the contract code.

Solidity version 0.8 changes

Version 0.8 introduces restrictions to explicit conversions and also requires `msg.sender` to be explicitly cast to `payable` in order to allow ether transfers [20]. This is also the first version with automatic checks for over- and underflow. Internal checks such as these are updated to use the `REVERT` opcode instead of the `INVALID` opcode.

7.4.3 Analysis approaches

To allow the tools to successfully detect the four reentrancy types, it is necessary for them to at least recognize `CALL`, `DELEGATECALL` and `CREATE` as potential causes for vulnerabilities. Mythril's "External Calls" module checks any `CALL`s made, while its "State Change External Calls" module triggers on `CALL`, `DELEGATECALL` and `CALLCODE` as well as `SSTORE`, `SLOAD`, `CREATE` and `CREATE2`. However, Mythril's analysis capability for delegated and create-based reentrancy is greatly reduced by the fact that in the case of receiving multiple contracts within a single Solidity source code file, it only analyzes one of the contracts [175].

Oyente's reentrancy checking function analyzes whether any `CALL`s found forward more gas than `send()` or `transfer()` would, and if so, flags the contract as reentrant. Based on both the published code and the authors' description, it seems that both `DELEGATECALL` and `CREATE` are not handled separately.

The authors of Sailfish state that they made the deliberate decision to not output any warnings should benign reentrancy be detected [65]; they define reentrancy to be a vulnerability only if it leads to a state inconsistency. The storage dependency graph

(SDG) that is built during Sailfish's first phase is queried for potential state inconsistencies, and only if such are found, does the analysis continue with the symbolic execution phase. However, both the usage of a graph as well as the restrictions put on the symbolic execution mean that there is a potential for oversimplification, which could lead to the tool missing a number of vulnerable contracts. The tool handles `DELEGATECALL` the same as other external calls, and checks for create-based reentrancy by combining both the parent and child contract into a single SDG.

As eThor's goal quite strongly differs from that of the other presented tools, a direct comparison may not actually be expedient. Instead, despite achieving comparatively low scores for accuracy and precision, the tool fulfilled its task and did not produce any false negatives. Not supporting `DELEGATECALL` has a strong impact on its overall results, as a third of the bytecode test data contains this specific opcode.

Except for Sailfish, none of the tools specifically target the lesser-known reentrancy types, which leads to lower overall results for cross-function, delegated, and create-based reentrancy. All symbolic execution tools are affected by the constraints imposed to avoid state space explosion, meaning that better results could be achieved by fine-tuning the available parameters. The results also show that the tools struggle with changes introduced by new Solidity versions and corresponding opcode updates, suggesting that recent developments and well-maintained tools should be preferred when selecting Smart Contract analysis tools. Furthermore, the selection of the tool should be tailored to the contract(s) to be analyzed, as the analysis performance is shown to be highly dependent on the properties of the input contract, i.e. the file type, Solidity version, and included opcodes.

CHAPTER 8

Summary and Conclusion

Reentrancy is a programming pattern which allows an Ethereum Smart Contract to be repeatedly called during a single transaction, possibly resulting in state inconsistencies and causing the loss of ether. In this thesis, we identify four different types of reentrancy: same-function, cross-function, delegated, and create-based reentrancy. While same-function reentrancy, the simplest form, allows an adversary to repeatedly re-enter a single function in a Smart Contract, cross-function reentrancy is triggered by entering two or more functions of the same contract. The more complex delegated reentrancy can occur when using Solidity's `delegatecall()` function, which passes control to an external contract and allows them to execute code within the context of the calling contract. Should the external contract make any unsafe external calls, this would impact the calling contract's state. The fourth type of reentrancy, create-based reentrancy, may happen during the creation of a new contract by another contract. Should the constructor of the new contract contain any unsafe calls, this could lead to a reentrancy attack against the original contract. We further identify two programming patterns that can be used to mitigate reentrancy vulnerabilities. The Checks-Effects-Interactions pattern requires the code to be arranged in a way so that state inconsistencies are avoided, whereas reentrancy locks introduce additional checks that fail upon detecting a reentrant call.

Such problematic code locations can be detected using Smart Contract analysis tools. We can categorize such tools into static and dynamic analysis tools, where static analysis tools do not require the execution of the code. Symbolic execution is a popular analysis method in which a concrete execution of the code is simulated and specific values are replaced by symbolic values, which may adhere to program-specific constraints. Symbolic execution suffers from the state space explosion problem, meaning that complex programs may require extensive amounts of resources to completely analyze all paths, making this approach impractical to use without additional constraints. Further approaches in static analysis include abstract interpretation as well as some Machine Learning techniques, whereas examples for dynamic analysis include fuzzing and runtime verification.

To gather information on the currently existing Smart Contract analysis tools and methods, we conduct a literature review. Excluding duplicates, we gather a total of 256 publications that were published from 2021 onward, 104 of which cover Smart Contract analysis for reentrancy detection. We briefly discuss the subset of tools that use a traditional static analysis approach, i.e. do not rely on Machine Learning techniques.

In order to identify the differences and similarities between Smart Contract analysis tools, we design an experimental study. To that end, we manually build a test data set containing both Solidity source code and bytecode, which covers all four reentrancy types and the Solidity versions 0.4 to 0.8. We furthermore select four static analysis tools, namely Oyente, Mythril, eThor, and Sailfish, which are executed on the test data. Our results show that all tools struggle with certain reentrancy types and may not be suited for reentrancy detection in practice. We consider both Oyente and eThor to be only suitable for the detection of same-function reentrancy, whereas Mythril and Sailfish may be used for the detection of two types each. Mythril manages to achieve sufficient scores for delegated reentrancy, while Sailfish does not consistently detect same-function reentrancy, but appears suitable for both cross-function reentrancy and create-based reentrancy. We further identify differences between the tools in regards to accuracy, precision, recall, sensitivity and error rate. Oyente tends to be overly lax, resulting in a number of false negatives and low recall. Mythril improves on recall compared to Oyente, but in turn generally suffers from lower precision and specificity. Our results confirm that eThor indeed manages to not produce any false negatives, which however leads to a high number of false positives and thus low accuracy, precision, and specificity. It further struggles with high error rates, partially caused by the fact that the tool does not support the `DELEGATECALL` opcode. Sailfish, too, is affected by a heightened error rate, which negatively impacts its overall accuracy. It does, however, manage to produce average results for Solidity version 0.4 and 0.5 contracts, where its error rate is comparatively low. The results show a negative correlation between Solidity version number and overall accuracy for both Oyente and Sailfish, which is likely caused by a lack of tool maintenance regarding new Solidity features and opcodes. The tools' differences are furthermore based on varying degrees of abstraction within the analysis process as well as the tools' different approaches as to what is considered a reentrancy vulnerability. Where Sailfish purposely attempts to detect cross-function, delegated, and create-based reentrancy, other tools focus on same-function reentrancy and monitor state changes in combination with `CALL`s.

Our results show that despite extensive research in the field, both recent and well-known tools may not be suitable for detecting reentrancy in practice. Instead, users may need to consult multiple tools or tailor their tool selection towards the properties of the tested contracts. Further research could expand the provided data set in order to provide a better balance between vulnerable and non-vulnerable contracts. The knowledge provided in this thesis should equip the reader with the knowledge to identify multiple types of reentrancy, allowing them to incorporate that knowledge into future work.

Overview of Generative AI Tools Used

The online tools DeepL[1] and ChatGPT[2] were used to improve the grammar and the spelling of individual sentences in the thesis.

[1] https://www.deepl.com
[2] https://chatgpt.com

List of Figures

List of Tables

Bibliography

[1] S. Tikhomirov, "Ethereum: State of Knowledge and Research Perspectives," in *Foundations and Practice of Security*, pp. 206–221, Springer International Publishing, 2018.

[2] M. Young, "Ethereum dominates among developers, but competitors are growing faster." `https://cointelegraph.com/news/ethereum-dominates-among-developers-but-competitors-growing-faster`, 2022. Accessed December 2022.

[3] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and Defenses," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–43, 2020.

[4] M. di Angelo and G. Salzer, "Automated Vulnerability Detection in Ethereum Bytecode: A Large Empirical Investigation," 2022. Submitted for publication.

[5] U. Mukhopadhyay, A. Skjellum, O. Hambolu, J. Oakley, L. Yu, and R. Brooks, "A brief survey of Cryptocurrency systems," in *2016 14th Annual Conference on Privacy, Security and Trust (PST)*, pp. 745–752, 2016.

[6] Bitcoin.org, "How does Bitcoin mining work?." `https://bitcoin.org/en/faq#how-does-bitcoin-mining-work`. Accessed April 2024.

[7] Ethereum.org, "Proof-of-stake (PoS)." `https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/`. Accessed April 2024.

[8] V. Buterin, "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform." `https://ethereum.org/en/whitepaper/`, 2014.

[9] Etherscan.io, "Ethereum gas tracker." `https://etherscan.io/gastracker`. Accessed April 2024.

[10] M. Pacheco, G. A. Oliva, G. K. Rajbahadur, and A. E. Hassan, "What makes Ethereum blockchain transactions be processed fast or slow? An empirical study," *Empirical Software Engineering*, vol. 28, no. 2, p. 39, 2023.

[11] Ethereum.org, "Gas and fees." https://ethereum.org/en/developers/docs/gas/. Accessed May 2024.

[12] Bitcoin.org, "Ethereum Virtual Machine (EVM)." https://ethereum.org/en/developers/docs/evm/. Accessed May 2024.

[13] M. di Angelo and G. Salzer, "Characterizing Types of Smart Contracts in the Ethereum Landscape," in *Financial Cryptography and Data Security* (M. Bernhard, A. Bracciali, L. J. Camp, S. Matsuo, A. Maurushat, P. B. Rønne, and M. Sala, eds.), pp. 389–404, Springer International Publishing, 2020.

[14] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger. Paris version 705168a," *Ethereum project yellow paper*, 2024. https://github.com/ethereum/yellowpaper.

[15] F. Heintel, "Solidity v0.1.0 turns 5! A walk down memory lane...," *soliditylang.org Blog*, 2024. https://soliditylang.org/blog/2020/07/08/solidity-turns-5/.

[16] nikola-matic, "Solidity GitHub releases." https://github.com/ethereum/solidity/releases. Accessed January 2025.

[17] soliditylang.org, "Language influences." https://docs.soliditylang.org/en/v0.8.26/language-influences.html. Accessed May 2024.

[18] soliditylang.org, "Layout of a Solidity Source File." https://docs.soliditylang.org/en/v0.8.26/layout-of-source-files.html. Accessed May 2024.

[19] soliditylang.org, "Contracts." https://docs.soliditylang.org/en/v0.8.26/contracts.html. Accessed May 2024.

[20] soliditylang.org, "Solidity v0.8.0 Breaking Changes." https://docs.soliditylang.org/en/v0.8.26/080-breaking-changes.html. Accessed May 2024.

[21] vyperlang.org, "Vyper." https://docs.vyperlang.org/en/v0.3.10/. Accessed May 2024.

[22] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks," 2018.

[23] National Institute of Standards and Technology, "Security Content Automation Protocol (SCAP) Version 1.3 Validation Program Test Requirements." https://doi.org/10.6028/NIST.IR.7511r5, 2018.

[24] Smart Contract Weakness Classification Registry, "SWC-107 Reentrancy." https://swcregistry.io/docs/SWC-107/. Accessed May 2024.

[25] soliditylang.org, "Security Considerations." https://docs.soliditylang.org/en/latest/security-considerations.html. Accessed May 2024.

[26] Smart Contract Security Field Guide, "Reentrancy." https://scsfg.io/hackers/reentrancy/. Accessed May 2024.

[27] soliditylang.org, "Units and Globally Available Variables." https://docs.soliditylang.org/en/latest/units-and-global-variables.html. Accessed June 2024.

[28] ConsenSys Diligence, "Stop Using Solidity's transfer() Now." https://consensys.io/diligence/blog/2019/09/stop-using-solioditys-transfer-now/. Accessed June 2024.

[29] uni-due-syssec, "Re-Entrancy Attack Patterns." https://github.com/uni-due-syssec/eth-reentrancy-attack-patterns. Accessed June 2024.

[30] S. Kim and S. Ryu, "Analysis of Blockchain Smart Contracts: Techniques and Insights," in *2020 IEEE Secure Development (SecDev)*, pp. 65–73, 2020.

[31] 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, 2022.

[32] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[33] Microsoft Research, "Z3." https://www.microsoft.com/en-us/research/project/z3-3/. Accessed August 2024.

[34] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A Versatile and Industrial-Strength SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Fisman and G. Rosu, eds.), pp. 415–442, Springer International Publishing, 2022.

[35] 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*, Association for Computing Machinery, 2016.

[36] B. Mueller, "Smashing Ethereum Smart Contracts for Fun and Real Profit," in *HITB SECCONF Amsterdam*, 2018.

[37] N. Veloso, "Conkas: A Modular and Static Analysis Tool for Ethereum Bytecode," 2021.

[38] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, Association for Computing Machinery, 1977.

[39] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: surviving out-of-gas conditions in Ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, 2018.

[40] 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.

[41] D. Ressi, A. Spanò, L. Benetollo, C. Piazza, M. Bugliesi, and S. Rossi, "Vulnerability Detection in Ethereum Smart Contracts via Machine Learning: A Qualitative Analysis," 2024.

[42] J. Cai, B. Li, J. Zhang, X. Sun, and B. Chen, "Combine sliced joint graph with graph neural networks for smart contract vulnerability detection," *Journal of Systems and Software*, vol. 195, p. 111550, 2023.

[43] W. J.-W. Tann, X. J. Han, S. S. Gupta, and Y.-S. Ong, "Towards Safer Smart Contracts: A Sequence Learning Approach to Detecting Security Threats," 2019.

[44] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding The Greedy, Prodigal, and Suicidal Contracts at Scale," 2018.

[45] T. H.-D. Huang, "Hunting the Ethereum Smart Contract: Color-inspired Inspection of Potential Attacks," 2018.

[46] T. Ball, "The concept of dynamic analysis," in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, p. 216–234, Springer-Verlag, 1999.

[47] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Computers & Security*, vol. 75, pp. 118–137, 2018.

[48] V. Wüstholz and M. Christakis, "Harvey: a greybox fuzzer for smart contracts," ESEC/FSE 2020, Association for Computing Machinery, 2020.

[49] Z. Liu, P. Qian, J. Yang, L. Liu, X. Xu, Q. He, and X. Zhang, "Rethinking Smart Contract Fuzzing: Fuzzing With Invocation Ordering and Important Branch Revisiting," *IEEE Transactions on Information Forensics and Security*, vol. 18, 2023.

[50] Y. Falcone, K. Havelund, and G. Reger, "A Tutorial on Runtime Verification," in *Engineering Dependable Software Systems*, 2013.

[51] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online Detection of Effectively Callback Free Objects with Applications to Smart Contracts," 2018.

[52] K. Sen, "Concolic testing," in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, p. 571–572, Association for Computing Machinery, 2007.

[53] 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," 2019.

[54] F. Marques, J. Fragoso Santos, N. Santos, and P. Adão, "Concolic Execution for WebAssembly (Artifact)," *Dagstuhl Artifacts Series*, vol. 8, no. 2, 2022.

[55] M. Barboni, A. Morichetta, and A. Polini, "SuMo: A mutation testing approach and tool for the Ethereum blockchain," *Journal of Systems and Software*, vol. 193, 2022.

[56] H. Rameder, "Systematic review of ethereum smart contract security vulnerabilities, analysis methods and tools," Master's thesis, TU Wien, 2021.

[57] M. Ren, F. Ma, Z. Yin, Y. Fu, H. Li, W. Chang, and Y. Jiang, "Making smart contract development more secure and easier," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Association for Computing Machinery, 2021.

[58] M. Ren, F. Ma, Z. Yin, H. Li, Y. Fu, T. Chen, and Y. Jiang, "SCStudio: a secure and efficient integrated development environment for smart contracts," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Association for Computing Machinery, 2021.

[59] Z. Ali Khan and A. Siami Namin, "Involuntary Transfer: A Vulnerability Pattern in Smart Contracts," *IEEE Access*, vol. 12, pp. 62459–62479, 2024.

[60] A. Alkhalifah, A. Ng, P. A. Watters, and A. S. M. Kayes, "A Mechanism to Detect and Prevent Ethereum Blockchain Smart Contract Reentrancy Attacks," *Frontiers in computer science (Lausanne)*, vol. 3, 2021.

[61] M. Almakhour, L. Sliman, A. E. Samhat, and A. Mellouk, "A formal verification approach for composite smart contracts security using fsm," *Journal of King Saud University - Computer and Information Sciences*, vol. 35, no. 1, 2023.

[62] N. Ashizawa, N. Yanai, J. P. Cruz, and S. Okamura, "Eth2Vec: Learning Contract-Wide Code Representations for Vulnerability Detection on Ethereum Smart Contracts," in *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*, Association for Computing Machinery, 2021.

[63] B. Boi, C. Esposito, and S. Lee, "Smart Contract Vulnerability Detection: The Role of Large Language Model (LLM)," *SIGAPP Applied Computing Review*, vol. 24, Aug. 2024.

[64] B. Boi, C. Esposito, and S. Lee, "VulnHunt-GPT: a Smart Contract vulnerabilities detector based on OpenAI chatGPT," in *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, Association for Computing Machinery, 2024.

[65] 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)*, 2022.

[66] C. Braghin, E. Riccobene, and S. Valentini, "Modeling and verification of smart contracts with Abstract State Machines," in *Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing*, Association for Computing Machinery, 2024.

[67] J. Cai, B. Li, T. Zhang, J. Zhang, and X. Sun, "Fine-grained smart contract vulnerability detection by heterogeneous code feature learning and automated dataset construction," *Journal of Systems and Software*, vol. 209, 2024.

[68] S. Cao, S. Dang, Y. Zhang, W. Wang, and N. Cheng, "A blockchain-based access control and intrusion detection framework for satellite communication systems," *Computer Communications*, vol. 172, 2021.

[69] Y. Chen, Z. Sun, Z. Gong, and D. Hao, "Improving smart contract security with contrastive learning-based vulnerability detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Association for Computing Machinery, 2024.

[70] J. Chen, C. Chen, J. Hu, J. Grundy, Y. Wang, T. Chen, and Z. Zheng, "Identifying Smart Contract Security Issues in Code Snippets from Stack Overflow," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, Association for Computing Machinery, 2024.

[71] D. Chen, L. Feng, Y. Fan, S. Shang, and Z. Wei, "Smart contract vulnerability detection based on semantic graph and residual graph convolutional networks with edge attention," *Journal of Systems and Software*, vol. 202, 2023.

[72] J. Cheng, Y. Chen, Y. Cao, and H. Wang, "A vulnerability detection framework by focusing on critical execution paths," *Information and Software Technology*, vol. 174, 2024.

[73] J. Cheng, Y. Chen, Y. Cao, and H. Wang, "A vulnerability detection framework with enhanced graph feature learning," *Journal of Systems and Software*, vol. 216, 2024.

[74] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "SMARTIAN: enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, IEEE Press, 2022.

[75] H. O. Demir, S. Z. Parlat, and A. Gumus, "Ethereum Blockchain Smart Contract Vulnerability Detection Using Deep Learning," in *2023 7th International Symposium on Innovative Approaches in Smart Technologies (ISAS)*, 2023.

[76] L. Dong and W. Li, "Reentrancy Vulnerability Detection Based on Semantic-Aware Graph Convolutional Neural Network," in *2023 3rd International Conference on Computer Science and Blockchain (CCSB)*, 2023.

[77] L. Duan, L. Yang, C. Liu, W. Ni, and W. Wang, "A New Smart Contract Anomaly Detection Method by Fusing Opcode and Source Code Features for Blockchain Services," *IEEE Transactions on Network and Service Management*, vol. 20, no. 4, 2023.

[78] M. Eshghie, C. Artho, and D. Gurov, "Dynamic Vulnerability Detection on Smart Contracts Using Machine Learning," in *Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering*, Association for Computing Machinery, 2021.

[79] Y. Fang, C. Wang, Z. Sun, and H. Cheng, "Jyane: Detecting Reentrancy vulnerabilities based on path profiling method," in *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, 2021.

[80] M. Feng, W. Mi, X. Zhang, B. Chen, and M. Huang, "A Smart Contract Vulnerability Detection Model Based on Multi-Type Features and Pre-Training Techniques," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, 2023.

[81] Z. Feng, Y. Feng, H. He, W. Zhang, and Y. Zhang, "A bytecode-based integrated detection and repair method for reentrancy vulnerabilities in smart contracts," *IET Blockchain*, vol. 4, no. 3, 2024.

[82] X. Feng, H. Liu, L. Wang, H. Zhu, and V. S. Sheng, "An interpretable model for large-scale smart contract vulnerability detection," *Blockchain: Research and Applications*, vol. 5, no. 3, 2024.

[83] C. Gao, W. Yang, J. Ye, Y. Xue, and J. Sun, "sGuard+: Machine Learning Guided Rule-Based Automated Vulnerability Repair on Smart Contracts," *ACM Transactions on Software Engineering and Methodology*, vol. 33, June 2024.

[84] I. Garfatta, K. Klai, M. Graïet, and W. Gaaloul, "Model checking of vulnerabilities in smart contracts: a solidity-to-CPN approach," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, Association for Computing Machinery, 2022.

[85] P. Gong, W. Yang, L. Wang, F. Wei, K. HaiLaTi, and Y. Liao, "GRATDet: Smart Contract Vulnerability Detector Based on Graph Representation and Transformer," *Computers, Materials and Continua*, vol. 76, no. 2, 2023.

[86] R. Guo, W. Chen, L. Zhang, G. Wang, and H. Chen, "Smart contract vulnerability detection model based on siamese network (scvsn): a case study of reentrancy vulnerability," *Energies*, vol. 15, no. 24, 2022.

[87] L. Guo, H. Huang, S. Xue, P. Wang, and L. Zhao, "Reentrancy Vulnerability Detection Based on Graph Convolutional Networks and Expert Patterns," in *2023 IEEE 16th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2023.

[88] L. Guo, H. Huang, L. Zhao, P. Wang, S. Jiang, and C. Su, "Reentrancy vulnerability detection based on graph convolutional networks and expert patterns under subspace mapping," *Computers & Security*, vol. 142, 2024.

[89] S. HajiHosseinKhani, A. H. Lashkari, and A. Mizani Oskui, "Unveiling vulnerable smart contracts: Toward profiling vulnerable smart contracts using genetic algorithm and generating benchmark dataset," *Blockchain: Research and Applications*, vol. 5, no. 1, 2024.

[90] L. Han, "Smart Contract Reentrancy Vulnerability Detection Based on CNN and LSTM-Attention," in *2024 5th International Seminar on Artificial Intelligence, Networking and Information Technology (AINIT)*, 2024.

[91] Y. He, H. Dong, H. Wu, and Q. Duan, "Formal Analysis of Reentrancy Vulnerabilities in Smart Contract Based on CPN," *Electronics*, vol. 12, no. 10, 2023.

[92] Z. He, Z. Zhao, K. Chen, and Y. Liu, "Smart Contract Vulnerability Detection Method Based on Feature Graph and Multiple Attention Mechanisms," *Computers, Materials and Continua*, vol. 79, no. 2, 2024.

[93] H. Huang, L. Guo, L. Zhao, H. Wang, C. Xu, and S. Jiang, "Effective combining source code and opcode for accurate vulnerability detection of smart contracts in edge ai systems," *Applied Soft Computing*, vol. 158, 2024.

[94] V. K. Jain and M. Tripathi, "Multi-Objective Approach for Detecting Vulnerabilities in Ethereum Smart Contracts," in *2023 International Conference on Emerging Trends in Networks and Computer Communications (ETNCC)*, 2023.

[95] H. Lakadawala, K. Dzigbede, and Y. Chen, "Detecting Reentrancy Vulnerability in Smart Contracts using Graph Convolution Networks," in *2024 IEEE 21st Consumer Communications & Networking Conference (CCNC)*, 2024.

[96] B. Le Hong, T. Le Duc, T. Doan Minh, D. Tran Tuan, D. Phan The, and H. Pham Van, "Contextual Language Model and Transfer Learning for Reentrancy Vulnerability Detection in Smart Contracts," in *Proceedings of the 12th*

International Symposium on Information and Communication Technology, ACM, 2023.

[97] B. Li, Z. Pan, and T. Hu, "ReDefender: Detecting Reentrancy Vulnerabilities in Smart Contracts Automatically," *IEEE Transactions on Reliability*, vol. 71, no. 2, 2022.

[98] W. Li, X. Gao, and R. Xu, "Reentrancy Vulnerability Detection Based on Conv1D-BiGRU and Expert Knowledge," in *2023 4th International Conference on Information Science, Parallel and Distributed Systems (ISPDS)*, 2023.

[99] Y. Li, R. Guo, G. Wang, L. Zhang, J. Qiu, S. Su, Y. Liu, G. Xu, and H. Chen, "An Efficient Detection Model for Smart Contract Reentrancy Vulnerabilities," in *Smart Computing and Communication*, vol. 13828 of *Lecture Notes in Computer Science*, Springer, 2023.

[100] L. Li, Y. Liu, G. Sun, and N. Li, "Smart Contract Vulnerability Detection Based on Automated Feature Extraction and Feature Interaction," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 9, 2024.

[101] B. Li, Z. Pan, and T. Hu, "EvoFuzzer: An Evolutionary Fuzzer for Detecting Reentrancy Vulnerability in Smart Contracts," *IEEE Transactions on Network Science and Engineering*, 2024.

[102] Z. Liu, P. Qian, X. Wang, Y. Zhuang, L. Qiu, and X. Wang, "Combining Graph Neural Networks With Expert Knowledge for Smart Contract Vulnerability Detection," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 2, 2023.

[103] H. Liu, Y. Fan, L. Feng, and Z. Wei, "Vulnerable smart contract function locating based on Multi-Relational Nested Graph Convolutional Network," *Journal of Systems and Software*, vol. 204, 2023.

[104] F. Luo and R. Luo, "VulDet: Smart Contract Vulnerability Detection Based on Graph Attention Networks," in *2023 2nd International Conference on Artificial Intelligence and Blockchain Technology (AIBT)*, 2023.

[105] Z. Luo, S. Chen, G. Wang, and H. Li, "Two-Stage Smart Contract Vulnerability Detection Combining Semantic Features and Graph Features," in *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2023.

[106] F. Luo, R. Luo, T. Chen, A. Qiao, Z. He, S. Song, Y. Jiang, and S. Li, "SCVHunter: Smart Contract Vulnerability Detection Based on Heterogeneous Graph Attention Network," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Association for Computing Machinery, 2024.

[107] A. Mittal, G. Widjaja, R. D. Cosme Pecho, R. Kiruba, J. M. Falcón Roque, and A. Chandra, "Blockchain Based Abstract Syntax Tree to Detect Vulnerability in IOT-Enabled Smart Contract," in *2023 Second International Conference On Smart Technologies For Smart Nation (SmartTechCon)*, 2023.

[108] K. L. Narayana and K. Sathiyamurthy, "Automation and smart materials in detecting smart contracts vulnerabilities in blockchain using deep learning," *Materials Today: Proceedings*, vol. 81, 2023. International Virtual Conference on Sustainable Materials (IVCSM-2k20).

[109] M. Ndiaye, T. A. Diallo, and K. Konate, "ADEFGuard: Anomaly detection framework based on Ethereum smart contracts behaviours," *Blockchain: Research and Applications*, vol. 4, no. 3, 2023.

[110] 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*, Association for Computing Machinery, 2022.

[111] S. B. Osei, Z. Ma, and R. Huang, "Smart contract vulnerability detection using wide and deep neural network," *Science of Computer Programming*, vol. 238, 2024.

[112] Z. Pan, T. Hu, C. Qian, and B. Li, "ReDefender: A Tool for Detecting Reentrancy Vulnerabilities in Smart Contracts Effectively," in *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, 2021.

[113] Y. Pan, Z. Xu, L. T. Li, Y. Yang, and M. Zhang, "Automated Generation of Security-Centric Descriptions for Smart Contract Bytecode," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, Association for Computing Machinery, 2023.

[114] M. Pasqua, A. Benini, F. Contro, M. Crosara, M. Dalla Preda, and M. Ceccato, "Enhancing Ethereum smart-contracts static analysis by computing a precise Control-Flow Graph of Ethereum bytecode," *Journal of Systems and Software*, vol. 200, 2023.

[115] B. Prasad and S. Ramachandram, "Prevention and Detection Mechanisms for Re-Entrancy Attack and King of Ether Throne Attack for Ethereum Smart Contracts," *Ingénierie des systèmes d'Information*, vol. 27, no. 5, 2022.

[116] P. Qian, Z. Liu, Y. Yin, and Q. He, "Cross-Modality Mutual Learning for Enhancing Smart Contract Vulnerability Detection on Bytecode," in *Proceedings of the ACM Web Conference 2023*, Association for Computing Machinery, 2023.

[117] S.-J. Qin, Z. Liu, F. Ren, and C. Tan, "Smart Contract Vulnerability Detection Based on Critical Combination Path and Deep Learning," in *Proceedings of the*

2022 12th International Conference on Communication and Network Security, Association for Computing Machinery, 2023.

[118] M. Rodler, D. Paaßen, W. Li, L. Bernhard, T. Holz, G. Karame, and L. Davi, "EF/CF: High Performance Smart Contract Fuzzing for Exploit Generation," in *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 2023.

[119] M. Rossini, M. Zichichi, and S. Ferretti, "Smart Contracts Vulnerability Classification through Deep Learning," in *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, Association for Computing Machinery, 2023.

[120] M. Sharma, *Optimizing Detection of Reentrancy attacks in Solidity Smart Contracts*. PhD thesis, Dublin, National College of Ireland, 2023.

[121] X. Shen and M. Li, "Smart Contract Reentrancy Vulnerability Detection Method Based on Deep Learning Hybrid Model," in *2023 5th International Conference on Artificial Intelligence and Computer Applications (ICAICA)*, 2023.

[122] C. Shou, S. Tan, and K. Sen, "ItyFuzz: Snapshot-Based Fuzzer for Smart Contract," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, Association for Computing Machinery, 2023.

[123] Y. Smaragdakis, N. Grech, S. Lagouvardos, K. Triantafyllou, and I. Tsatiris, "Symbolic value-flow static analysis: deep, precise, complete modeling of ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 5, Oct. 2021.

[124] G. Sun, C. Jiang, J. Shen, and Y. Zhang, "SCOBERT: A Pre-Trained BERT for Smart Contract Vulnerability Detection," in *2023 8th International Conference on Data Science in Cyberspace (DSC)*, 2023.

[125] X. Sun, L. Tu, J. Zhang, J. Cai, B. Li, and Y. Wang, "ASSBert: Active and semi-supervised bert for smart contract vulnerability detection," *Journal of Information Security and Applications*, vol. 73, 2023.

[126] U. Tahir, F. Siyal, M. Ianni, A. Guzzo, and G. Fortino, "Exploiting Bytecode Analysis for Reentrancy Vulnerability Detection in Ethereum Smart Contracts," in *2023 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/-CyberSciTech)*, 2023.

[127] X. Tang, Y. Du, A. Lai, Z. Zhang, and L. Shi, "Deep learning-based solution for smart contract vulnerabilities detection," *Scientific Reports*, vol. 13, p. 20106, Nov. 2023.

[128] V. Tong, C. Dao, T. Dong, H. A. Tran, D. Tran, and T. X. Tran, "POSTER: Multi-Block Fusion Mechanism for Multi-label Vulnerability Detection in Smart Contracts," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, Association for Computing Machinery, 2024.

[129] T.-D. Tran, K. A. Vo, D. T. Phan, C. N. Tan, and V.-H. Pham, "ChainSniper: A Machine Learning Approach for Auditing Cross-Chain Smart Contracts," in *Proceedings of the 2024 9th International Conference on Intelligent Information Technology*, Association for Computing Machinery, 2024.

[130] Y. Wang, J. Zhao, Y. Zhang, X. Hei, and L. Zhu, "Smart Contract Symbol Execution Vulnerability Detection Method Based on CFG Path Pruning," in *Proceedings of the 5th ACM International Symposium on Blockchain and Secure Critical Infrastructure*, Association for Computing Machinery, 2023.

[131] Z. Wang, J. Chen, Y. Wang, Y. Zhang, W. Zhang, and Z. Zheng, "Efficiently Detecting Reentrancy Vulnerabilities in Complex Smart Contracts," *Proceedings of the ACM on Software Engineering*, vol. 1, July 2024.

[132] D. Wang, M. Shan, and N. Tong, "Smart Contract Vulnerability Detection Based on Machine Learning," in *2024 6th International Conference on Electronic Engineering and Informatics (EEI)*, 2024.

[133] D. Wang, J. Chen, S. Cai, Q. Feng, Y. Chen, and X. Hu, "SCVD-SA: A Smart Contract Vulnerability Detection Method Based on Hybrid Deep Learning Model and Self-attention Mechanism," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, 2024.

[134] Z. Wang, J. Chen, P. Zheng, Y. Zhang, W. Zhang, and Z. Zheng, "Unity is Strength: Enhancing Precision in Reentrancy Vulnerability Detection of Smart Contract Analysis Tools," *IEEE Transactions on Software Engineering*, 2024.

[135] Z. Wang, W. Dai, M. Li, K.-K. R. Choo, and D. Zou, "DFier: A directed vulnerability verifier for Ethereum smart contracts," *Journal of Network and Computer Applications*, vol. 231, 2024.

[136] H. Wu, Z. Zhang, S. Wang, Y. Lei, B. Lin, Y. Qin, H. Zhang, and X. Mao, "Peculiar: Smart Contract Vulnerability Detection Based on Crucial Data Flow Graph and Pre-training Techniques," in *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*, pp. 378–389, 2021.

[137] G. Xu, L. Liu, and Z. Zhou, "Reentrancy Vulnerability Detection of Smart Contract Based on Bidirectional Sequential Neural Network with Hierarchical Attention Mechanism," in *2022 International Conference on Blockchain Technology and Information Security (ICBCTIS)*, 2022.

[138] G. Xu, L. Liu, and J. Dong, "Vulnerability Detection of Ethereum Smart Contract Based on SolBERT-BiGRU-Attention Hybrid Neural Model," *CMES - Computer Modeling in Engineering and Sciences*, vol. 137, no. 1, 2023.

[139] X. Xu, J. Xu, Y. Fu, and W. Tian, "Security Analysis of UAV Swarm Based on Smart Contracts," in *2024 International Conference on Intelligent Computing and Robotics (ICICR)*, 2024.

[140] Y. Xue, M. Ma, Y. Lin, Y. Sui, J. Ye, and T. Peng, "Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, Association for Computing Machinery, 2021.

[141] X. Yan, S. Wang, and K. Gai, "A Semantic Analysis-Based Method for Smart Contract Vulnerability," in *2022 IEEE 8th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, 2022.

[142] S. Yang, J. Chen, and Z. Zheng, "Definition and Detection of Defects in NFT Smart Contracts," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, Association for Computing Machinery, 2023.

[143] S. Yang, J. Chen, M. Huang, Z. Zheng, and Y. Huang, "Uncover the Premeditated Attacks: Detecting Exploitable Reentrancy Vulnerabilities by Identifying Attacker Contracts," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, 2024.

[144] H. Yang, X. Gu, X. Chen, L. Zheng, and Z. Cui, "CrossFuzz: Cross-contract fuzzing for smart contract vulnerability detection," *Science of Computer Programming*, vol. 234, 2024.

[145] J. Ye, M. Ma, Y. Lin, L. Ma, Y. Xue, and J. Zhao, "Vulpedia: Detecting vulnerable ethereum smart contracts via abstracted vulnerability signatures," *Journal of Systems and Software*, vol. 192, 2022.

[146] M. Ye, Y. Nan, H.-N. Dai, S. Yang, X. Luo, and Z. Zheng, "FunFuzz: A Function-Oriented Fuzzer for Smart Contract Vulnerability Detection with High Effectiveness and Efficiency," *ACM Transactions on Software Engineering and Methodology*, vol. 33, Sept. 2024.

[147] M. Ye, X. Lin, Y. Nan, J. Wu, and Z. Zheng, "Midas: Mining Profitable Exploits in On-Chain Smart Contracts via Feedback-Driven Fuzzing and Differential Analysis," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, Association for Computing Machinery, 2024.

[148] T. Yin, C. Zhang, Y. Ni, Y. Wu, T. Wong, X. Luo, Z. Li, and Y. Guo, "An empirical study on implicit constraints in smart contract static analysis," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, Association for Computing Machinery, 2022.

[149] X. Yu, H. Zhao, B. Hou, Z. Ying, and B. Wu, "DeeSCVHunter: A Deep Learning-Based Framework for Smart Contract Vulnerability Detection," in *2021 International Joint Conference on Neural Networks (IJCNN)*, 2021.

[150] R. Yu, J. Shu, D. Yan, and X. Jia, "ReDetect: Reentrancy Vulnerability Detection in Smart Contracts with High Accuracy," in *2021 17th International Conference on Mobility, Sensing and Networking (MSN)*, 2021.

[151] L. Yu, J. Lu, X. Liu, L. Yang, F. Zhang, and J. Ma, "PSCVFinder: A Prompt-Tuning Based Framework for Smart Contract Vulnerability Detection," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, 2023.

[152] D. Yuan, X. Wang, Y. Li, and T. Zhang, "Optimizing smart contract vulnerability detection via multi-modality code and entropy embedding," *Journal of Systems and Software*, vol. 202, 2023.

[153] S. Zeng, R. Chen, H. Zhang, and J. Wang, "A High-Performance Smart Contract Vulnerability Detection Scheme Based on BERT," in *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, 2023.

[154] L. Zhang, J. Wang, W. Wang, Z. Jin, Y. Su, and H. Chen, "Smart contract vulnerability detection combined with multi-objective detection," *Computer Networks*, vol. 217, 2022.

[155] H. Zhang, W. Zhang, Y. Feng, and Y. Liu, "SVScanner: Detecting smart contract vulnerabilities via deep semantic extraction," *Journal of Information Security and Applications*, vol. 75, 2023.

[156] L. Zhang, Y. Li, R. Guo, G. Wang, J. Qiu, S. Su, Y. Liu, G. Xu, H. Chen, and Z. Tian, "A Novel Smart Contract Reentrancy Vulnerability Detection Model based on BiGAS," *Journal of signal processing systems*, vol. 96, no. 3, 2024.

[157] K. Zhou, J. Huang, H. Han, B. Gong, A. Xiong, W. Wang, and Q. Wu, "Smart contracts vulnerability detection model based on adversarial multi-task learning," *Journal of Information Security and Applications*, vol. 77, 2023.

[158] H. Zhu, K. Yang, L. Wang, Z. Xu, and V. S. Sheng, "GraBit: A Sequential Model-Based Framework for Smart Contract Vulnerability Detection," in *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, 2023.

[159] K. Zkik, A. Sebbar, O. Fadi, O. Mustapha, and A. Belhadi, "A Graph Neural Network Approach for Detecting Smart Contract Anomalies in Collaborative Economy Platforms Based on Blockchain Technology," in *2023 9th International Conference on Control, Decision and Information Technologies (CoDIT)*, 2023.

[160] souffle-lang.github.io, "Welcome - Soufflé." `https://souffle-lang.github.io/docs.html`. Accessed November 2024.

[161] usyd-blockchain, "Vandal." `https://github.com/usyd-blockchain/vandal`. Accessed October 2024.

[162] J. Feist, G. Grieco, and A. Groce, "Slither: A Static Analysis Framework for Smart Contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, p. 8–15, 2019.

[163] crytic, "Slither." `https://github.com/crytic/slither`. Accessed October 2024.

[164] enzymefinance, "Oyente." `https://github.com/enzymefinance/oyente`. Accessed October 2024.

[165] Consensys, "Mythril." `https://github.com/Consensys/mythril`. Accessed October 2024.

[166] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "eThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts," 2020.

[167] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Network and Distributed System Security Symposium*, 2018.

[168] Security & Privacy TU Wien, "eThor." `https://secpriv.wien/ethor`. Accessed October 2024.

[169] Y. Tang, Z. Li, and Y. Bai, "Rethinking of Reentrancy on the Ethereum," in *2021 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/-CyberSciTech)*, 2021.

[170] 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*, p. 664–676, Association for Computing Machinery, 2018.

[171] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006. ROC Analysis in Pattern Recognition.

[172] soliditylang.org, "Solidity v0.5.0 Breaking Changes." `https://docs.soliditylang.org/en/latest/050-breaking-changes.html`. Accessed January 2025.

[173] soliditylang.org, "Solidity v0.6.0 Breaking Changes." https://docs.soliditylang.org/en/latest/060-breaking-changes.html. Accessed January 2025.

[174] soliditylang.org, "Solidity v0.7.0 Breaking Changes." https://docs.soliditylang.org/en/latest/070-breaking-changes.html. Accessed January 2025.

[175] Mythril documentation, "Security Analysis." https://mythril-classic.readthedocs.io/en/master/security-analysis.html. Accessed January 2025.