Investigating Vulnerability Detection Tools and Solidity Compiler Versions on the Vulnerability of Unchecked Call Return Values

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Theresa Bodner, BSc.
Matrikelnummer 11807865

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 1. September 2025

Theresa Bodner | Gernot Salzer

TU WIEN Informatics

Investigating Vulnerability Detection Tools and Solidity Compiler Versions on the Vulnerability of Unchecked Call Return Values

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering and Internet Computing

by

Theresa Bodner, BSc.
Registration Number 11807865

to the Faculty of Informatics

at the TU Wien

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

Vienna, September 1, 2025

Theresa Bodner	Gernot Salzer

Erklärung zur Verfassung der Arbeit

Theresa Bodner, 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, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 1. September 2025

Theresa Bodner

Danksagung

Mein besonderer Dank gilt Gernot und Monika, die mich nicht nur fachlich hevorragend betreut haben, sondern auch immer motivierend zur Seite standen und Zeit für mich hatten. Es war mir eine Freude, mit Euch diese Arbeit zu verfassen und mein Studium abzuschließen.
Danke den viele Kolleginnen und Kollegen, die ich im Laufe der Jahre kennenlernen durfte und mit denen ich schöne wie herausfordernde Zeiten gemeinsam erlebt habe.
Danke meiner Familie, die mich immer unterstützt und begleitet und mir diese Studienzeit ermöglicht hat.

Kurzfassung

Blockchain-Technologien haben in den letzten Jahren erheblich an Aufmerksamkeit gewonnen. Ethereum hat sich als führende Plattform für Smart Contracts etabliert. Während Smart Contracts eine Vielzahl von Anwendungen in Bereichen wie dezentralisierte Finanzen, Non-Fungible Tokens und dezentralisierte autonome Organisationen ermöglichen, bringen sie zugleich neue Herausforderungen mit sich, auch im Bereich Security. Schwachstellen in bereits deployten Smart Contracts können schwerwiegende Folgen haben, unter anderem finanzielle Verluste. Da bereits auf der Blockchain deployte Smart Contracts schwer zu verändern sind, sind zuverlässige Tools zur Erkennung von Schwachstellen besonders wichtig.

Diese Arbeit untersucht die Schwachstelle der "unchecked return values of low-level calls" und deren Varianten. Low-Level Operationen wie `send`, `call`, `delegatecall`, `staticcall` und `callcode` führen bei einem Fehler kein Revert aus, sondern geben einen booleschen Wert zurück, der überprüft werden muss. Zur Analyse dieser Schwachstelle entwickeln wir ein Framework zur Generierung von Minimal Contracts mit der Schwachstelle sowohl in Konstruktoren als auch in Funktionen und kompiliert mit verschiedenen Solidity Compiler Versionen von 0.4 bis 0.8.

Mit diesem Datensatz an 595 Minimal Contracts evaluieren wir drei Werkzeuge zur Erkennung von Schwachstellen: Mythril und Slither basieren auf statischer Analyse, während ConFuzzius ein dynamisches Analysetool auf Basis von Hybrid-Fuzzing ist.

Die Ergebnisse zeigen Unterschiede in den Erkennungsfähigkeiten. Mythril zeigt die ausgewogenste Performanz über Low-Level Konstrukte und Compiler Versionen hinweg, scheitert jedoch bei call-basierten Contracts mit Solidity Version 0.8. Slither erkennt die meisten Varianten konsistent, übersieht jedoch eine Gruppe von Testfällen. ConFuzzius erzielt gute Ergebnisse bei `send` und `call`, kann jedoch Schwachstellen in `delegatecall`, `staticcall` und `callcode` nicht identifizieren, ebenso wenig wie Schwachstellen in Konstruktoren, da nur Runtime Code analysiert wird.
Insgesamt liefert diese Studie wertvolle Einblicke in die Stärken und Schwächen aktueller Tools zur Schwachstellenerkennung in Smart Contracts.

Abstract

Blockchain technologies have gained significant attention in recent years, with Ethereum establishing itself as a leading platform for smart contracts. While enabling a wide range of applications in areas such as decentralised finance, non-fungible tokens, and decentralised autonomous organisations, smart contracts also introduce new security challenges. Vulnerabilities in deployed contracts can have severe consequences, including financial losses. Since contracts on the blockchain are difficult to modify, reliable vulnerability detection tools to support developers are of great importance.

This thesis investigates the weakness of "unchecked return values of low-level calls" and its variants. Low-level operations such as `send`, `call`, `delegatecall`, `staticcall`, and `callcode` do not revert upon failure but instead return a boolean value which must be verified. To analyse this weakness, we develop a framework for generating minimal contracts with varied weakness placements, including both constructors and functions, across Solidity compiler versions from 0.4 to 0.8.

With this data set of 595 minimal contracts, we evaluate three vulnerability detection tools: Mythril and Slither, which employ static analysis, and ConFuzzius, a hybrid fuzzing-based dynamic analysis tool.

The results highlight differences in the detection capabilities. Mythril demonstrates the most balanced performance across low-level constructs and compiler versions, but fails on call-based contracts with Solidity compiler version 0.8. Slither detects most variants consistently, yet misses certain cases. ConFuzzius performs well on `send` and `call` but is unable to detect weaknesses in `delegatecall`, `staticcall`, and `callcode` and does not analyse weaknesses placed in constructors due to its runtime-only approach.
Overall, this study provides valuable insight into the strengths and limitations of current vulnerability detection tools for smart contracts.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

In recent years, blockchain technologies have gained significant interest, with Ethereum emerging as a leading platform for smart contracts. Beyond digital wallets and cryptocurrency exchanges, blockchain and smart contracts enable a wide range of new applications in the field of decentralised finance (DeFi), non-fungible tokens (NFTs) for ownership, decentralised autonomous organisations (DAOs), supply chain management for transparency, and many more.

Despite its popularity and new possibilities, smart contract technology is still in its infancy and introduces new challenges as well as security risks which must be taken very seriously. Vulnerable or faulty smart contracts can be exploited leading to severe consequences, such as financial loss, loss of trust, and general decline in trust in the technology.

Once deployed on the blockchain, a smart contract is difficult to modify. Therefore, we need to put time and effort into the development of smart contracts and focus on security aspects, especially with financial assets at stake. Fortunately, there are numerous tools at hand that support developers in this task of finding potential weaknesses. However, most tools are over- or under-reporting (often depending on the detection approach), and even a combination of these tools does not guarantee the detection of a certain vulnerability. There can even be discrepancies among tools indicating caution regarding reported findings. A study by di Angelo et al. [dADFS23] compares the results of 12 vulnerability scanners on a data set of Ethereum contracts from 6 years and shows that "there is little agreement between the tools regarding the findings, even for well-researched and frequently analyzed weaknesses [...]".

To improve such detection tools, a deep understanding of vulnerabilities and all their variants is needed. Several types of vulnerabilities prevalent in smart contracts have been discovered and categorised into taxonomies [VIL24a], [RAS22], [swc18]. Some of these vulnerabilities have already been studied to a greater extent, such as the reentrancy attack [1], while others are still relatively unexplored.

In this work, we pick one weakness: the weakness of "unchecked return values of low-level calls" and its variants. Smart contracts can invoke other contracts, and the call can fail and revert. Low-level methods, like `call` or `send`, do not revert. Instead, they return `false` upon failure which must be checked when returning from an external call. Neglecting to do this check can corrupt the subsequent computation and lead to errors. Moreover, malicious actors may exploit this by intentionally causing the external call to fail.

We focus on one weakness and examine whether a set of selected detection tools can consistently identify all variants of this weakness, or whether discrepancies arise – both between different variants within a single tool and across the tools. This knowledge should help researchers and developers of both smart contracts and detection tools better understand the weakness and improve the effectiveness as well as the accuracy of detection tools.

1.2 Research Questions

To guide the study, we formulate the following research questions.

RQ1: **Which variants of the weakness of unchecked call return values exist?** We are interested in how different taxonomies define the weakness and in which variants it can appear.

RQ2: **How did error handling in smart contracts evolve?** We aim to explore how Solidity implements error handling on a language level and investigate whether proper utilisation of these mechanisms can mitigate the weakness.

RQ3: **To what extent can inconsistencies or discrepancies be identified in the results of vulnerability detection tools?** Do the tools handle a specific variant of the weakness in an anomalous way? This question will guide the experimental study of vulnerability scanners, the results of which we aim to analyse and compare.

RQ4: **To what extent can varying results be identified across different Solidity compiler versions?** By using different Solidity compiler versions, we want to study how compiler updates might affect bytecode generation and whether these differences have an impact on the vulnerability detection.

1.3 Methodological Approach

By answering the research questions stated above, we want to systematically analyse vulnerability detection tools and Solidity compiler versions and their behaviour towards

[1] https://swcregistry.io/docs/SWC-107/

variations of unchecked call return values. We conduct a literature review to find variants of the weakness and compare existing taxonomies of vulnerability classifications.

For the experimental study, we develop a benchmark data set of smart contracts written in Solidity representing the identified variants. The benchmark contracts only focus on the core properties of the weakness and thus represent so-called *minimal contracts*. We evaluate a selected number of detection tools on the data set, allowing us to examine how each tool handles the weakness, and whether they produce varying or even contradicting results. The minimal contracts are compiled using different Solidity compiler versions, and we analyse if compiler updates have an influence on the tools' detection ability.

1.4 Related Work

Di Angelo et al. [AMS24] systematically studied integer bugs and implemented a benchmark set with varying parameters to test vulnerability scanners and profile Solidity compiler versions. The results reveal discrepancies and inconsistencies in tool behaviour, while variants of the bug have been eliminated with updated Solidity versions.

Fürst [Fü25] focusses in her diploma thesis on the reentrancy attack. The study consolidates existing research by categorising reentrancy types, analysing Solidity coding patterns for prevention, and evaluating smart contract analysis tools. With a hand-crafted data set, four tools and different Solidity compiler versions are assessed. The results indicate that not all tools can handle all types of the vulnerability. More complex types are only detected by individual tools. The findings suggest the use of multiple tools for better security analysis.

Durieux et al. [DFAC20] conducted an extensive study on nine detection tools using two data sets: a set of annotated vulnerable cmart contracts and a collection of smart contracts from Etherscan. They found that only 42% of annotated vulnerable smart contracts were detected by all tools, whereas 97% of real-world contracts were marked as vulnerable, indicating a high false-positive rate.

Kado et al. [KYCO23] examine the impact of Solidity compiler updates on vulnerabilities and focus on three specific issues – including the unchecked call vulnerability – across approximately 503,000 contracts sourced from Etherscan. Their findings indicate that compiler updates generally contribute to a reduction in vulnerabilities. In the case of the unchecked call vulnerability, its occurrence has decreased in Solidity version 0.8, however, it remains still high due to false positives and code cloning.

1.5 Structure of the Thesis

The structure of the thesis is as follows. Chapter 2 provides background information on blockchain basics, Ethereum, smart contracts, and Solidity. In chapter 3 we introduce the weakness of unchecked low-level call return values as defined in different taxonomies.

Chapter 4 explains vulnerability detection techniques which are used by common vulnerability scanners. The methodology for the practical part of this thesis is described in chapter 5 and the results presented in chapter 6 before concluding and summarising in chapter 7.

CHAPTER 2

Background

This chapter provides background information on blockchain technology and serves as a short introduction to Ethereum, smart contracts, and Solidity with its characteristics relevant to the practical part of this thesis.

2.1 Blockchain

Blockchain is a distributed ledger technology that enables the secure recording of transactions across a decentralised network without relying on a central authority. Yaga et al. [YMRS19] provide an excellent overview of the fundamental concepts of blockchain technology. The basic idea is to model it as a state transition machine. There is a state which holds the information such as ownership of digital assets or account balances. Transactions serve as inputs to a transition function that processes the current state and a transaction to produce a new state.

Once a transaction is published on the blockchain, it becomes tamper-proof. This comes from the design of the blockchain database, where one block is a structured unit containing the validated transactions, a timestamp, a cryptographic hash of the previous block, and a unique hash of its own, among other entries. Linking each block to its predecessor allows for a secure and chronological chain which ensures that it cannot be altered without changing all subsequent blocks. The validation of transactions comes from the group of participants that collectively agree on the validity through a consensus protocol.

This system based on cryptographic mechanisms to secure data and validate ownership makes it particularly suitable for electronic cash applications, the first of which was Bitcoin in 2008. In addition to Bitcoin, Ethereum is another cryptocurrency network, which was introduced in 2014.

2.2 Ethereum

Ethereum is a decentralised blockchain platform proposed by Buterin [But14] in 2013 with the possibility to run programs – so-called *smart contracts* – on. This allows anyone to write decentralised applications and to define their own transition functions. Information is passed between *accounts* identified by a 20-byte address. There are two types of accounts on the network: Externally owned accounts are user-controlled accounts that can initiate transactions and interact with smart contracts. Contract accounts hold code and execute logic when invoked. For each transaction a fee has to be paid in Ether, Ethereum's currency.

2.2.1 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is the runtime environment that executes smart contracts on the Ethereum blockchain. It is a stack-based machine and features several types of storage. Among them is the *stack* which holds local variables or return values; the *memory* holds transient data during contract execution; the contract's *long term storage* is persistent and saves all contract variables and mappings.

Each smart contract is compiled into EVM bytecode which consists of EVM opcodes (operation codes) that perform basic stack instructions like `ADD`, `SUB`, `AND`, `OR`, ..., as well as blockchain specific operations [evm25]. The deployment code, or contract creation code, includes instructions for the construction of the contract which is executed only once during the deployment and is responsible for initialising contract variables, as seen in listing 2.1. The runtime code is permanently saved at the contract's address and is executed when invoked.

2.2.2 Smart Contracts

Smart contracts are programs deployed on the blockchain that are invoked through transactions sent by externally owned accounts and message calls from other contracts. Once triggered, they execute code based on specified conditions and rules. They function as self-enforcing agreements between parties, where no intermediator is needed to verify the execution [int25]. The term smart contract refers to Nick Szabo [Sza94], who introduced the concept of a "computerized transaction protocol that executes the terms of a contract" in 1994.

The most popular programming language for smart contracts on the Ethereum network is Solidity, which was initiated by Gavin Wood in 2015.

2.2.3 Solidity

Solidity is a high-level object-oriented programming language for smart contracts that is influenced by languages like C++, Python, and JavaScript [sol25b]. Listing 2.1 presents the fundamental elements of a smart contract demonstrating a simple Ether deposit and withdrawal mechanism.

A contract is similar to a class in object-oriented languages. It holds state variables like `owner`, mappings (`investorsBalance`), enums, or structs, which can be modified through functions. These functions can be called from outside the contract (public visibility) or can be hidden (private visibility).

The Solidity compiler version is defined via the `pragma` directive in line 1. We can (but do not have to) implement a `constructor` (line 8) which holds the initialisation code and is executed only once upon deployment. In the example, the variable `owner` is initialised with the address of the account that deployed the contract (`msg.sender`). `investorsBalance` is a mapping which is a key-value pair. In the example, an Ethereum address is mapped to an integer value in line 5 which tracks how much Ether each user has deposited.

The keyword `payable` (line 13) is needed to declare that the function `deposit()` can receive Ether. The function `withdraw()` (line 19) allows the investors to withdraw all their funds stored and it resets the balance of the investor back to 0 (line 23).

The `require` keyword is used to enforce a condition. If the condition is not passed, the execution halts and the transaction is reverted. In the SimpleContract in line 14 we see, that the Ether value we call the function with must be greater than or equal to 1 Ether. Line 21 ensures that the investor has actually Ether deposited. Line 24 shows the transfer of Ether to the sender's address. Explicitly casting the address `msg.sender` to payable ensures that Ether can be sent. The `transfer()` function allows to send the specified amount of Ether and automatically reverts on failure.

Listing 2.1: Example presenting basic Solidity language elements

```
1  pragma solidity 0.8.29;
2
3  contract SimpleContract {
4      address public owner;
5      mapping(address => uint256) public investorsBalance;
6
7      // Constructor to set the owner
8      constructor() {
9          owner = msg.sender;
10     }
11
12     // Payable function to receive Ether and update mapping
13     function deposit() public payable {
14         require(msg.value >= 1 ether, "Must send Ether");
15         investorsBalance[msg.sender] += msg.value;
16     }
17
18     // Withdraw function: investors can withdraw their funds
19     function withdraw() public {
```

```
        uint256 amount = investorsBalance[msg.sender];
        require(amount > 0, "No funds to withdraw");

        investorsBalance[msg.sender] = 0;
        payable(msg.sender).transfer(amount);
    }
}
```

Low-Level Language Constructs

Solidity offers different mechanisms for interacting with other contracts and transferring Ether. The following low-level methods are members of the type `address` and offer developers flexibility and more direct control, which also comes with additional risks and requires careful handling. They typically return a boolean variable indicating success or failure, along with any returned data [sol25b].

- **`send`** is the low-level counterpart to the `transfer` function as seen in the example in listing 2.1. It is used to exchange Ether. In case of failure, `send` does not revert, but returns `false`.
- **`call`** is used to call external contracts and invoke their functions or send Ether. The computations affect the state of the called contract.
- **`delegatecall`** is also used to execute code from another contract, with the difference that any state changes are made on the calling contract.
- **`staticcall`**, introduced in version 0.5.0, can be used in the same way as `call`, but to invoke read-only functions. `staticcall` reverts if any attempt is made to modify the state.
- **`callcode`** has been removed in Solidity version 0.5.0, but worked in a similar way, however, without access to the original `msg.sender` and `msg.value` values.

Error Handling

Solidity provides structured error handling, described in the documentation [sol25a]. Especially in smart contract development, error handling is of high importance, as contracts often control operations of monetary value, and weaknesses can lead to significant financial loss if exploited.

Generally, state-reverting exceptions are used, which means that all changes to the state within the transaction are undone, and the error is propagated to the caller. Exceptions occurring in sub-calls are automatically propagated and rethrown, unless caught via `try/catch`. There are two built-in types of errors: `Error(string)` is used for regular error conditions and `Panic(uint256)` signals errors that should never happen in a properly functioning contract.

The functions `assert` and `require` are used to validate conditions and trigger exceptions when those conditions are not satisfied. The documentation suggests using `assert` only for internal errors and the check of invariants, as it creates an error of type `Panic`. `require` can be used to ensure programming conditions and throws an error of type `Error`. Both of these control structures internally trigger a `revert` operation which has the same functionality as the old keyword `throw` (removed in Solidity version 0.5.0) and can also be explicitly called. `assert`, `require`, and `revert` were introduced in Solidity version 0.4.10.

The `try/catch` statement offers the possibility of catching failures in external function calls. If the external call succeeds, the `try` block executes and continues with code execution after the `catch` block, similarly to other programming languages.

The low-level constructs differ in the way they handle errors from the "standard" way of error handling in Solidity. `send`, `call`, `delegatecall`, and `staticcall` never throw an exception, but return `false` in the case of failure, which has to be checked by developers upon return. Additionally, it is necessary to manually verify the existence of the target account before making the call.

Mitropoulos et al. [MKM+24] investigate the development of error handling in Solidity and its usage by analysing 283.000 unique Solidity smart contracts from version 0.1.2 to 0.8.19. They discover that error handling features have frequently been changing across Solidity versions, typically being updated once or twice per year on average. The `require` statement is by far the most used feature for evaluating function arguments and external calls with about 83%, while `assert` is the least used with almost 4%. `require` as a Solidity-specific feature also demonstrates the highest usage increase across Solidity versions. Moreover, the authors provide categories and frequencies of error handling misuse and reveal that for about 63% of external calls neither `require` nor `try/catch` was used. They explicitly mention the highest number of error handling misuse over time in the context of checking the success of external contract calls.

CHAPTER 3

Unchecked Low-Level Call Return Value

This chapter lays the foundation for the vulnerability analysis conducted in this thesis by explaining basic concepts like weakness and vulnerability, before diving into definitions of the weakness in focus. Finally, we address mitigation strategies for developers to avoid or minimise the risk of introducing the weakness in their code.

3.1 Weakness and Vulnerability

The Common Weakness Enumeration (CWE) is a comprehensive and community-maintained list that serves to standardise the classification of software and hardware weaknesses. Developed with contributions from government, industry, and academia, CWE helps security professionals and developers understand the underlying causes of security issues. One of its contributions is the distinction it draws between weaknesses and vulnerabilities:

A **weakness** is described as a condition or design flaw in software, firmware, hardware, or a service component that can, under certain circumstances, lead to a security issue [cweb].

A **vulnerability**, on the other hand, is the actual flaw that results from such a weakness and can be exploited by hackers, for example, to cause a breach in confidentiality, integrity, or availability [cwea].

As in all areas of software development, it is important to identify weaknesses early on and address them at their root to prevent them from evolving into serious vulnerabilities.

Many papers in the literature use these terms interchangeably. We want to make the distinction as proposed by the CWE.

In the context of Ethereum smart contracts, a wide range of weaknesses have been identified, many of which have been systematically categorised into taxonomies. Vidal et al. [VIL24a], for example, speak of 481 types of vulnerabilities. Some of these vulnerabilities are more general in nature and also occur in traditional software systems, such as integer overflows/underflows, while others are unique to the smart contract domain, like vulnerabilities concerning transaction fees. Although certain issues have already been studied in the literature, this study focusses on one particular weakness: unchecked return values from low-level calls and its variants.

3.2 Unchecked low-level call return value

At this time, this weakness is still a prevalent issue in smart contracts. For example, OWASP (Open Worldwide Application Security Project) elevated it from position 10 in its 2023 ranking to position 6 in the 2025 list, highlighting its relevance [owa25].
Di Angelo et al. [dADFS23] identified this weakness as the second most frequent issue in their data set of over 48 million Ethereum contracts deployed over six years, ranking just after reentrancy and ahead of integer overflows and underflows.

A possible consequence of the weakness is the continuation of contract execution under false assumptions. For example, if a low-level call that is meant to transfer funds fails silently and the balance is still updated in the contract, the system may proceed in a corrupted state which leads to financial inconsistencies or exploitation.

In the following, we take a look at taxonomies and how the weakness is defined, as well as possible mitigations and implementation recommendations.

3.2.1 Taxonomies & Definitions

SWC Registry

The SWC Registry is a community-based project launched in 2018 with the goal of providing a way to classify security issues and defining a common language for describing these issues. The registry comprises 37 vulnerabilities; however, since 2020 it is no longer maintained.

SWC-104: Unchecked Call Return Value

> “The return value of a message call is not checked. Execution will resume even if the called contract throws an exception. If the call fails accidentally or an attacker forces the call to fail, this may cause unexpected behaviour in the subsequent program logic.” [swc]

In the thesis, we use the term “SWC-104” as abbreviation for the weakness studied.

OpenSCV

OpenSCV [VIL24a] is a hierarchical taxonomy for smart contract vulnerabilities. It was launched in 2024 with the goal of updating existing classifications and addressing issues such as arbitrary nomenclatures, concept inconsistencies, and inadequate granularity. The taxonomy has been assessed through a survey of experts in the field of smart contract security and is open to community contribution.

1.3 Improper Check of External Call Result

> "This category groups vulnerabilities that verify the execution of external contracts in an improper manner (i.e., verification is wrong or even missing), which affects the subsequent logic of the calling contract. The result of invoking a certain external operation should be verified, first of all, because it may simply fail, but especially because the called operation may be malicious (or may just have been poorly coded, resulting in an unexpected result); thus, the direct use of the result may lead to unexpected behavior."[VIL24a, p.30]

1.3.3 Improper Check of Low-Level Call Return Value

> "Languages like Solidity provide the possibility of using low-level calls that operate over raw addresses. Such calls do not verify that the code exists or the success of the calls. Thus, its use may lead to unexpected behavior [...]. As a result, using such calls can be risky and should be avoided in most cases."[VIL24a, p.30]

OWASP 10

OWASP 10 is a global foundation with the goal of improving the security of software. It is known for providing resources, guidelines, and creating standards for researchers, developers, and security professionals. In the OWASP Smart Contract Top 10 the weakness comes in 6th place.

Unchecked External Calls

> "Unchecked external calls refer to a security flaw where a contract makes an external call to another contract or address without properly checking the outcome of that call. In Ethereum, when a contract calls another contract, the called contract can fail silently without throwing an exception. If the calling contract doesn't check the return value, it might incorrectly assume the call was successful, even if it wasn't. This can lead to inconsistencies in the contract state and vulnerabilities that attackers can exploit." [owa25]

Literature review and taxonomy by Rameder

In 2021, Rameder [RAS22] conducted an extensive, systematic literature review on smart contract vulnerabilities, detection tools, and methods. The novel taxonomy comprises 54 vulnerabilities classified into 10 groups based on the DASP TOP 10 collection, the SWC Registry, and 17 selected surveys.

3A Unchecked low level call/send return value

> “Certain low level calls do not raise an exception when they fail. Instead the boolean false is returned. Missing to check and handle the return value adequately can lead to critical vulnerabilities.” [RAS22][p.53]

> “Certain low level functions, such as send, call, delegatecall and callcode do not throw exceptions if they fail and therefore require the developer to check the success of the operation.”[RAS22][p.62]

3.2.2 Example and Mitigation

The following contract in listing 3.1 serves as an illustrative example of the weakness. While the first function `callunchecked` (line 5) contains the weakness, the second function `callchecked` (line 9) checks the return value through the `require` construct.

Listing 3.1: Example of vulnerability and its safe variant

```
1  pragma solidity 0.8.29;
2
3  contract ReturnValue {
4
5    function callunchecked(address callee) public {
6      callee.call("");
7    }
8
9    function callchecked(address callee) public {
10     (bool success, ) = callee.call("");
11     require(success);
12   }
13 }
```

The example shows that with the addition of the `require` statement the risk can be reduced.

For safer Ether transfer, the Solidity documentation suggests using the `transfer` function instead of `send` which automatically reverts on failure. Regarding call-based constructs, it is explicitly stated that these functions “should be used with care”. [sol25b]

Verheijke and Rocha [VR22] conducted an exploratory study on the usage of specific Solidity language constructs in a data set of almost 27.000 smart contracts from July 2021 to January 2022. Around 55% of the contracts are of Solidity version 0.8.x. They investigate the number of occurrences of constructs like `call`, `transfer`, `send`, and guards like `require`, `assert`, and `revert`.

From the data set they analyse how many Ether exchange methods and guards, like `require`, are used in these contracts. The `call` function is used in 50% of the contracts, followed by the safer variant `transfer` with 34%. Regarding guard features, `require` is used almost in all contracts (97%). This high usage and popularity of the feature is also confirmed by Mitropoulos et al. [MKM+24], described in section 2.2.3.

CHAPTER 4

Vulnerability Detection Techniques

This chapter describes various techniques used by vulnerability scanners to detect security flaws in smart contracts. We take a broad view of the classifications of detection methods available before discussing some techniques in more detail as they are used by the tools we choose for the practical part of this study.

Rameder [RAS22] proposes a comprehensive taxonomy that categorises properties and methods of security analysis tools into seven main categories: input, main goals, analysis type, code transformation methods, formal methods, dynamic methods, and other or heuristic methods, which we will briefly summarise.

The **input type** determines whether the tools operate on bytecode or high-level source code. Their **main goals** vary, ranging from vulnerability detection and program correctness to gas/resource analysis, bulk or fully automated analysis, exploit generation, and other goals. The **analysis type** can be divided into static analysis, which examines code without execution, and dynamic analysis, which observes behaviour during runtime. There are different **code transformation methods** such as control flow graphs, data-flow and transaction analysis and execution traces, abstract syntax trees, decompilation, intermediate representation languages, disassembly, and finite state machines. **Formal methods** provide mathematical analysis through techniques such as symbolic execution, formal verification and theorem proving, constraint and SMT solving, among other methods. In the category of **dynamic methods**, fuzzing, runtime verification, concolic testing, and mutation testing explore programs during execution. Finally, **other or heuristic methods** include pattern matching or syntactical analysis, code instrumentation, machine learning, and taint analysis for vulnerability detection.

Vidal et al. [VIL24b] review the literature on detection techniques and categorise them together with a mapping to common vulnerability classification schemes. Their taxonomy comprises four categories: formal methods, code analysis, software testing, and machine learning. In the category of **formal methods** fall methods such as abstract interpretation, model checking, symbolic execution, and theorem proving. **Code analysis** include taint analysis and pattern recognition. Fuzzing and concolic execution are part of **software testing**. **Machine learning** consists of techniques such as classical machine learning, deep learning, and ensemble learning.

These categories provide an introduction and broad overview of common techniques. Tools often combine multiple approaches to balance out limitations of a single method. In the following sections, we have a closer look at the approaches employed by Mythril, Slither, and ConFuzzius.

4.1 Symbolic Execution

Detection approaches in the category of formal methods apply computational logic to verify whether a system's behaviour meets specific properties or sets of properties defined in a formal specification. The process begins with specifications and assumptions, which are formally modelled into verifiable properties. These properties serve as criteria for correctness, and the formal model of the system is systematically checked against them. The result is either that the model satisfies the specified properties – confirming correctness – or it fails – indicating an error.

Baldoni et al. [BCD+18] provide an overview of symbolic execution techniques, key aspects, as well as challenges.
Symbolic execution systematically explores numerous possible execution paths of a contract without relying on concrete input values, but instead using symbolic inputs. This is in contrast to a single concrete input which would only explore one specific control flow path. Each decision point in the code, such as an `if` statement, creates a new branch in the execution flow. These branches are captured in a Control Flow Graph, which models the possible paths the program can take, containing the propositional formula with the constraints that must be satisfied for that path to be executed. The propositional formulas are examined by automated reasoning tools, so-called Satisfiability Modulo Theories (SMT) solvers such as Z3, which check the satisfiability of the formulas.

As programs grow in size and complexity, symbolic execution inevitably encounters the well-known state explosion problem, where the number of possible execution paths increases, making the process time- and space-intensive. However, in the context of smart contract analysis, this challenge is less severe. Smart contracts are typically small, self-contained programs, and the EVM is less complex than other computing environments. This makes symbolic execution well-suited for analysing smart contracts and it may even achieve 100% code coverage [Mue18].

Gorgoris [Gor21][p.27] presents the basic principle of symbolic execution with the following example as illustrated in listing 4.1.

Listing 4.1: Simple program for symbolic execution example

```
def compare (x: int, y: int) -> int:
    if (x != y):
        if (x > y):
            return 1
        else:
            return -1
    return 0
```

The program takes two integers, compares them, and returns an integer value. The control flow graph and the symbolic execution tree are depicted in figure 4.1 and 4.2.

The program takes two inputs, x and y, which are provided externally. These variables are initialised with the symbolic values $x : S_x$ and $y : S_y$ as seen at the root of the symbolic execution tree. The constraints encountered along each path are shown in the nodes below the symbols. At the leaves of the tree we see possible sets of constraints at the end of the execution of the program: $\{S_x \neq S_y, S_x > S_y\}$, $\{S_x \neq S_y, S_x \leq S_y\}$, and $\{S_x = S_y\}$. These sets can be transformed into propositional formulas and solved with an SMT solver.

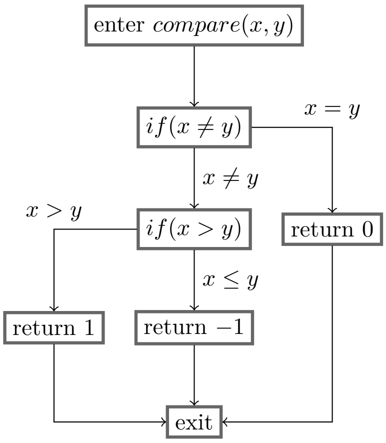

Figure 4.1: Control Flow Graph

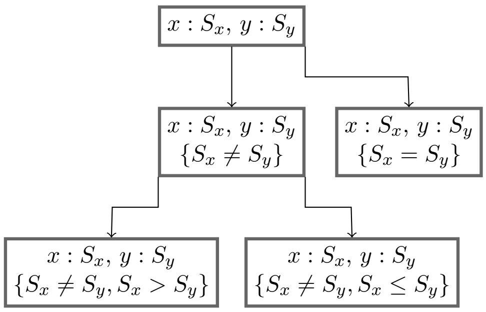

Figure 4.2: Symbolic Execution Tree

4.2 Taint Analysis

Taint analysis is a technique that can be employed in a static and dynamic way. The general idea, as described in the survey by Vidal et al. [VIL24b], is to trace the information flow and determine if a potentially dangerous (tainted) input can affect the program execution in a negative way. Especially user input can be a source of harmful data and can be traced whether it influences sensitive operations. Values are annotated with markings and followed while they get propagated through the program. The static approach does not require the setup of a blockchain environment by modelling all possible execution paths. The dynamic variant also involves running the contract, which leads to additional runtime information about the contract's behaviour.

Gorgoris [Gor21] describes the dynamic taint analysis:
The main components are the taint source, propagation rules, and the taint sink. The taint source is the origin of untrusted and potentially malicious data, for example, function parameters, data from users, or data returned from external calls. The values of these sources are marked as tainted.
The propagation rules describe how the taint spreads through the program. If a condition depends on a tainted variable, the taint can influence control-flow decisions. If any operand in a computation is tainted, the result is also tainted.
The taint sink is a sensitive operation, where the presence of tainted data could cause a security risk. That can be writing storage variables, interaction with other contracts with user-controlled destinations, or something similar.

4.3 Fuzzing

A dynamic approach to vulnerability detection involves testing during runtime through automatically generated input data, so-called fuzzing. These malformed or unexpected data help to detect bugs which might not be found through conventional test case generation. For example, a smart contract is executed with a large number of random inputs while being monitored for abnormal behaviour, which can indicate a vulnerability. While black-box fuzzing tests a program without any knowledge of its internal structure, grey-box fuzzing uses partial information to guide the input generation.

As Torres et al. [TIGS21] describe, an advanced variant is hybrid fuzzing, which combines fuzzing with symbolic execution. The analysis process starts with fuzzing until no additional code coverage is achieved after a predefined number of execution steps, then switches to symbolic execution to explore uncovered branching conditions. After solving them, the testing is continued in fuzzing mode.

4.4 Concolic Testing

Concolic testing combines the two complementary approaches of concrete and symbolic execution of a program. Wang et al. [WSC+18] describe concolic testing and investigate the optimal strategy, which also includes the question of when to switch from concrete to symbolic execution.

Concrete execution is computationally cheap; it runs the program with sampled real inputs, which is efficient for exploring paths that are easy to reach. However, it is more difficult to detect low-probability paths that are unlikely to be triggered by these inputs. In contrast, symbolic execution determines the constraints required to execute specific paths, which is useful for targeting rare branching conditions. It is, however,

computationally more expensive.
In concolic testing, symbolic execution is applied to program parts with low execution probability, while concrete execution handles the rest of the program, ideally achieving high testing coverage.

CHAPTER 5

Experiment Design

To answer the research questions RQ3 and RQ4 from section 1.2, we carry out an experiment to study the differences or even discrepancies between detection tools, as well as the impact of Solidity compiler versions. A selection of vulnerability scanners is run on a data set of smart contracts developed for this study. This chapter describes the setup for the experiment, the smart contract data set, the selection of detection tools, and the metrics used for evaluating the results.

5.1 Minimal Contracts data set

To represent a broad variety of the SWC-104 weakness, we decide to design the data set ourselves. We do not want to reuse existing data sets since these sets do not cover all variants we want to test and tend to be too complex for our use case. This study concentrates exclusively on evaluating the tools' capability to detect the weakness in its most fundamental form. Therefore, we omit complex smart contract logic and design a set of what we refer to as *minimal contracts*.

As described in section 2.2.3, the contracts are built around the low-level language constructs `send`, `call`, `delegatecall`, `staticcall` and `callcode`. We use templating with Mustache [1] to facilitate and automate the contract generation process. Since the usage of `send` differs slightly from the other low-level constructs, we differentiate between *send-based* and *call-based* contracts, for which we design two separate Mustache templates. In addition to these two variants, we test if the tools can detect the weakness when it appears in function code versus during deployment in the constructor, for which we again use two templates.

As an example, listing 5.1 shows the template for call-based contracts with the potential vulnerability placed in the function code. During the contract generation process, the

[1] https://github.com/janl/mustache.js/

placeholders are replaced by concrete values from a separate `data.json` file. This `data.json` file holds all possible values for the placeholders. First, all variations are generated and in a next step sorted out by constraints, as some combinations are not valid.

Listing 5.1: Mustache template for call-based contracts

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity {{ compilerVersion }};

{{#targetContract}}
{{{ targetContract }}}
{{/targetContract}}

contract {{ contractName }} {

    {{#variable}}
    {{{ variable }}}
    {{/variable}}

    constructor() public {

    }

    function test(address callAddress) public
    {{#returnSuccess}}
    {{{ returnSuccess }}}
    {{/returnSuccess}}{

        {{{success}}}callAddress.{{{ lowLevelConstruct }}}({{{payload}}});

        {{#checkSuccess}}
        {{{ checkSuccess }}}
        {{/checkSuccess}}

        {{#assignVariable}}
        {{{ assignVariable }}}
        {{/assignVariable}}
    }
}
```

For the placeholders, the following values are considered together with constraints:

(This serves more as an explanation and illustration of the process; it is not an exhaustive list, nor is it 1:1 applicable to the template.)

- `compilerVersions`: Solidity versions 0.4.26, 0.5.17, 0.6.12, 0.7.6, 0.8.29
- `targetContract`: implemented target contract with function `foo()` or `null`

- `variable`: `uint256 public publicVar;` or `null`
- `returnSuccess`: `returns (bool)` or `null`
- `success`:
  - `(bool success, )` for Solidity versions ≥ 0.5.17
  - `bool success` only for Solidity version 0.4.26
  - `null` for the case we do not want to check the return value, to create vulnerable contracts
- `lowLevelConstruct`: `send`, `call`, `delegatecall`, `staticcall` (for Solidity versions ≥ 0.5.17), `callcode` (only for Solidity version 0.4.26)
- `payload`:
  - `abi.encodeWithSignature("foo()")` for Solidity versions ≥ 0.5.17
  - `bytes4(keccak256("foo()")` for Solidity version 0.4.26
- `checkSuccess`: only if we have previously assigned the return value to a success variable
  - `require(success);`
  - `if (success) {} else { revert ("Error"); }`
  - `return success;` This variant is included to reflect the perspective that propagating the success status constitutes a form of return value handling, similar, for example, to a proxy contract, where we want to directly forward values without modification. Detection tools could interpret this in different ways. Logically, this variant appears only in contracts where the weakness is located within a function.
  - `null` if we do not want to check for the return value, as in vulnerable contracts
- `assignVariable`: `publicVar = 42;` assignment after low-level call, only if we have introduced the variable `publicVar` before, otherwise `null`

This leaves us with 315 contracts testing function code and 280 contracts testing deployment code – in total 595 minimal contracts. Of these 595 contracts, 280 contain a weakness. Table 5.1 shows the number of contracts per Solidity version and low-level construct.

The templates, data, contract render scripts, and all files mentioned in the following can be found in a public thesis repository: `https://github.com/TheresaBo/minimal_contracts_104`

	0.4.26	0.5.17	0.6.12	0.7.6	0.8.29
send	17	17	17	17	17
call	34	34	34	34	34
delegatecall	34	34	34	34	34
staticcall	-	34	34	34	34
callcode	34	-	-	-	-

Table 5.1: Number of minimal contracts per low-level construct and Solidity version

5.2 Compiler Versions

As briefly mentioned in section 5.1, we compile the contracts with five different Solidity compiler versions. The programming language Solidity is still evolving, with features being added and removed. Moreover, different compiler versions may generate different code patterns for the same source code, partly driven by changes in the EVM. Changes in source code and bytecode may affect the results of vulnerability detection. Therefore, we focus on the latest compiler versions before the next major release, which results in versions 0.4.26, 0.5.17, 0.6.12, 0.7.6, and 0.8.29 (cut-off date 28.04.2025).

5.3 SmartBugs 2.0

SmartBugs, developed by di Angelo et al. [ADFS23], is an open source execution framework for weakness detection that supports developers in testing their smart contracts for possible vulnerabilities. The framework allows for the integration of different vulnerability scanners using Docker. It offers a homogeneous interface and result reports for easier execution and comparison. It automates experiment execution and collects structured results in CSV files, which makes it useful for benchmarking, research, and tool development in Ethereum smart contract security. Today, SmartBugs comprises 20 tools that work on Solidity source code, byte code, as well as runtime code. Additionally, SmartBugs detects the correct compiler version stated in the contract. The modular framework also allows for easy integration of new tools or updates of existing tools. Because of these advantages and the possibility of reproducibility, we decided to use SmartBugs 2.0.10 [2] for the analysis of our minimal contracts.

5.4 Tool Selection

To ensure a meaningful and representative evaluation, the selection of tools was guided by some criteria. Needless to say, the tools have to support the detection of SWC-104 and cover a board range of Solidity compiler versions (0.4.x to 0.8.x). Ideally, they are already integrated into the SmartBugs framework, which also implies public availability.

[2]https://github.com/smartbugs/smartbugs

We prioritise tools that are actively maintained, well documented, and offer developer support.

We restricted the selection to three tools: Mythril, Slither, and ConFuzzius, with Mythril and Slither being state-of-the-art static analysis tools and ConFuzzius being a dynamic tool taking a hybrid fuzzing approach.

Mythril is a detection tool by Mueller [Mue18] and consists of different modules. Each module is tailored to detect specific types of weaknesses; the detection of SWC-104 relies on the module "Unchecked Retval". Mythril uses techniques such as symbolic execution and taint analysis to explore potential execution paths. It is an open-source project backed by the ConsenSys community [3] and actively maintained.

Slither is a static analysis framework by Feist et al. [FGG19] first published in 2018, that converts Solidity code into an intermediate representation, called SlithIR, and applies a range of detectors to uncover security issues. The relevant detectors for SWC-104 are "Unchecked low-level calls" and "Unchecked Send". It is openly available on GitHub [4].

An updated version of **ConFuzzius**, a dynamic analysis tool based on fuzzing, was added to SmartBugs with a new Docker image for this study. Torres et al. [TIGS21] take a hybrid approach; they combine symbolic execution with fuzzing and leverage dynamic data dependency analysis. The relevant detector for the studied weakness is "unchecked_return_value" and can also be found on GitHub [5].
However, since ConFuzzius only analyses runtime code and executes the contract using several fuzzed inputs, it does not generate results for contracts where the weakness has been placed in the constructor.

5.5 Workflow

We outline the workflow for evaluating the vulnerability detection tools. The process begins by structuring the variants and designing minimal contracts to enable a systematic analysis. As detailed in section 5.1, these contracts are generated automatically using a templating approach. They are then compiled to obtain the corresponding bytecode and opcode, which are used for further inspection and validation.

Next, the chosen detection tools are executed on this data set, and their outputs are parsed into CSV files – a feature provided by SmartBugs. To identify the occurrence of the SWC-104 vulnerability, we scan these CSV files for specific indicators: For Mythril, we look for `Unchecked_return_value_from_external_call_SWC_104`; Slither reports `unchecked_lowlevel` or `unchecked_send`; and ConFuzzius flags `Unhandled_Exception`. Based on these data, we extract true positives, false positives, true negatives, and false negatives (more detailed in section 5.6) for further analysis and calculation of evaluation metrics.

[3] https://github.com/ConsenSysDiligence/mythril
[4] https://github.com/crytic/slither
[5] https://github.com/christoftorres/ConFuzzius

5.6 Metrics

Classification metrics are essential tools for evaluating the performance of detection or prediction models. In our context, they help quantify how well the tools correctly identify software weaknesses. Metrics provide different perspectives on the correctness and completeness of the results. By applying these metrics, we can systematically compare detection tools, identify their strengths and weaknesses, and ensure that the analysis is based on measurable, interpretable outcomes rather than subjective impressions. Tharwat [Tha21] provides an excellent overview of these classification methods.

From the result files, we extract the following values:

- True Positive: a vulnerability is detected in a vulnerable contract
- False Positive: a vulnerability is reported in a safe contract
- True Negative: no vulnerability is detected in a safe contract
- False Negative: no vulnerability is reported for a vulnerable contract

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

Table 5.2: Confusion matrix

With these statistical values, the following metrics can be computed, where TP represents the absolute number of true positive cases – respectively TN, FP, and FN.

Accuracy

Accuracy measures the overall proportion of correctly classified contracts, including both safe and vulnerable ones:

$$Accurracy = \frac{TP + TN}{TP + TN + FP + FN}$$

Recall

Recall, also known as the true positive rate, measures the proportion of correctly identified vulnerable (positive) contracts among all actual vulnerable contracts:

$$Recall = \frac{TP}{TP + FN}$$

Precision

Precision measures the proportion of positive samples that were correctly classified out of all samples that were predicted as positive:

$$Precision = \frac{TP}{FP + TP}$$

Specificity

Specificity, also known as the true negative rate, measures the ratio of correctly classified negative (safe) contracts to the total number of actual negative contracts:

$$Specificity = \frac{TN}{FP + TN}$$

F1-Score

With precision and recall, the F_1-Score can be computed, which is the harmonic mean of precision and recall that balances false positives and false negatives. A high F_1-Score indicates a high classification performance.

$$F_1\text{-}Score = \frac{2 * Precision * Recall}{Precision + Recall}$$

Moreover, we are interested in the overlap between the result sets of the detection tools and compute their similarity by using the **Jaccard index**. This value provides a simple and computationally efficient way to compare sets, making it well-suited for our primary use case. A and B are sets of flagged contracts. We can compute their bidirectional similarity with the formula:

$$Jaccard = \frac{|A \cap B|}{|A \cup B|}$$

To determine the unidirectional overlap – or interiority – of the result sets, we compute

$$Overlap = \frac{|A \cap B|}{|A|}$$

and, respectively,

$$Overlap = \frac{|A \cap B|}{|B|}$$

In other words, we want to measure how much of set A is contained in set B, and vice versa. An overlap of 100% from A to B means that every element of A is also contained in set B. If an overlap of 100% is also observed in the reverse direction, then the two sets contain the same elements.

The Jaccard index and interiority, as well as other generalisations, are described in [dFC21].

CHAPTER 6

Evaluation and Result Interpretation

In this chapter, we present and analyse the results obtained from our experiments. The primary goal is to identify similarities and differences between the evaluated detection tools. To do so, we examine the calculated metrics as introduced in Chapter 5. The results are grouped and compared across different dimensions, including the detection tool, Solidity compiler version, and other contract characteristics. This evaluation aims to provide a deeper understanding of the behaviour of each tool and the limitations in detecting weaknesses.

To ensure the reliability and consistency of our results, we executed the experiment runs twice. This repetition aimed to rule out any non-deterministic behaviour or transient issues that might affect the detection tools. Both runs produced identical results, confirming the reproducibility of the tool outputs. Furthermore, this consistency indicates that the presence or absence of the weakness is deterministic within our data set.

The result data were collected from the parsed CSV files in SmartBugs and further analysed in Microsoft Excel, which can be found in `results_analysed_numbers.xlsx`.

6.1 Calculated Metrics

In the following, we organise the results by different categories. Since ConFuzzius does not analyse constructor code, comparisons for function contracts are made across all three tools, while constructor contracts are only compared between Mythril and Slither.

6.1.1 Overall Results

The evaluation results reveal notable differences in performance among the three tools. We start by discussing the overall results for function contracts shown in table 6.1 and

	Accuracy	Precision	Recall	Specificity	F_1-Score
Mythril	0,83	0,8	0,83	0,83	0,81
Slither	0,78	1	0,5	1	0,67
ConFuzzius	0,7	0,8	0,43	0,91	0,56

Table 6.1: Overall results for three tested detection tools for **function** contracts

	Accuracy	Precision	Recall	Specificity	F_1-Score
Mythril	0,91	1	0,83	1	0,91
Slither	0,75	1	0,5	1	0,67

Table 6.2: Overall results for two tested detection tools for **constructor** contracts

constructor contracts in table 6.2.

Mythril shows the most balanced performance, with all metrics above 80%. This reveals that Mythril correctly detects weaknesses while minimising false positives. The high F_1-Score additionally indicates that it performs well for function contracts as well as constructor contracts.

The results for constructor contracts may appear better at first glance; however, in these cases, a category of test cases – those involving the variant of `return success` – is absent. Such cases only occur when a function explicitly returns the success status, which cannot appear in contracts where we test the constructor.

Slither demonstrates perfect precision and specificity in function and constructor contracts, both 100%, which means that it does not make false positive classifications. Its recall with 50% shows that it misses a significant number of actual weaknesses. Interestingly, Slither does not report a weakness when the return value of a low-level call is assigned to a success variable but is subsequently not checked, shown in listing 6.1. This constitutes 140 contracts, which is half of all vulnerable contracts.

Listing 6.1: Code snippet of a variant without check, not flagged by Slither

```
(bool success, ) =
    callAddress.call(abi.encodeWithSignature("foo()"));
```

ConFuzzius has an accuracy of 70% among function contracts. Its recall is low with around 43%, meaning that there are difficulties in flagging contracts with weaknesses. With an F_1-Score of 56% it lies behind Mythril and Slither. We see the reasons for these numbers in section 6.1.2.

For a graphic comparison, figure 6.1 shows the overall F_1-Score of the three tools for function contracts and constructor contracts for Mythril and Slither.

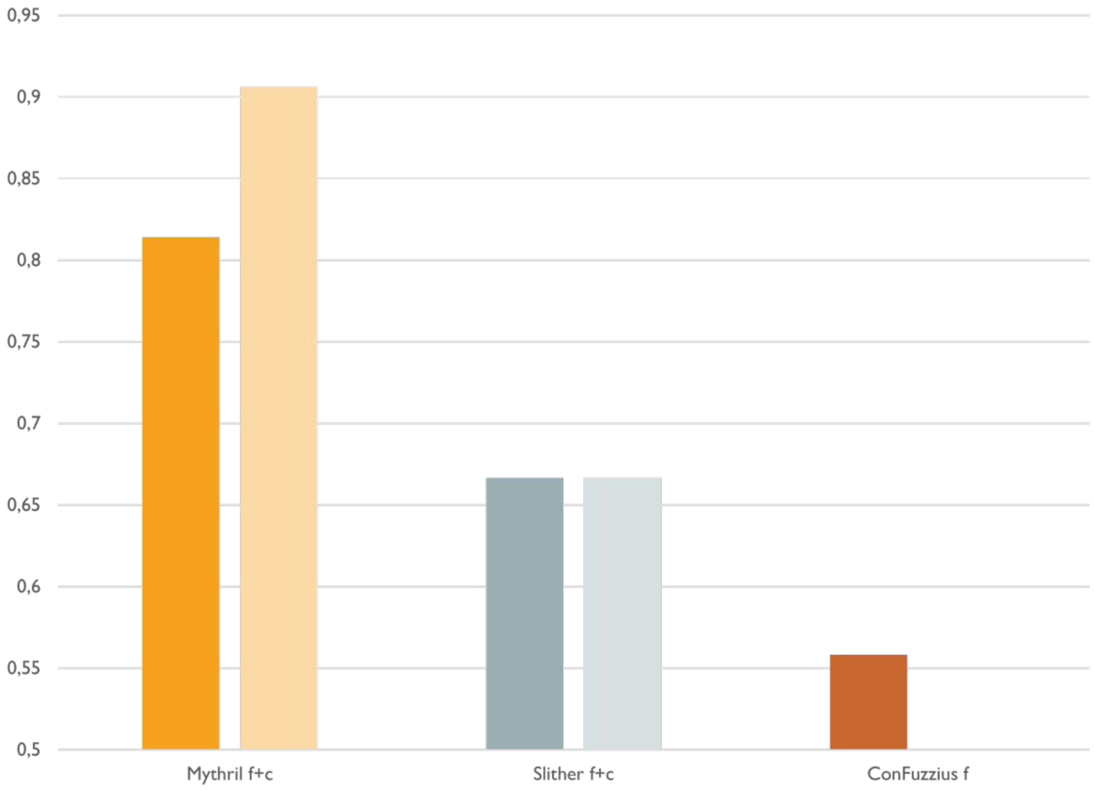

Figure 6.1: Comparison of overall F_1-Score

MYTHRIL function	**Accuracy**	**Precision**	**Recall**	**Specificity**	**F_1-Score**
send	0,89	0,8	1	0,8	0,89
call	0,82	0,8	0,8	0,84	0,8
delegatecall	0,82	0,8	0,8	0,84	0,8
staticcall	0,81	0,8	0,75	0,85	0,77
callcode	0,89	0,8	1	0,8	0,89

Table 6.3: Results by low-level constructs for Mythril **function** contracts

6.1.2 Results by Low-Level Construct

To gain deeper insight, we split the results into the different low-level constructs per tool and again distinguish between function and constructor contracts.

Mythril offers the most balanced and reliable detection across all low-level types as well as function contracts, table 6.3, and constructor contracts, table 6.4, with high values among all metrics.

Similarly to the overall results, **Slither** achieves perfect precision and specificity, but misses half of the actual weaknesses, hence the low recall values, which arise from one

MYTHRIL constructor	**Accuracy**	**Precision**	**Recall**	**Specificity**	**F_1-Score**
send	1	1	1	1	1
call	0,9	1	0,8	1	0,89
delegatecall	0,9	1	0,8	1	0,89
staticcall	0,88	1	0,75	1	0,86
callcode	1	1	1	1	1

Table 6.4: Results by low-level constructs for Mythril **constructor** contracts

SLITHER function	**Accuracy**	**Precision**	**Recall**	**Specificity**	**F_1-Score**
send	0,78	1	0,5	1	0,67
call	0,78	1	0,5	1	0,67
delegatecall	0,78	1	0,5	1	0,67
staticcall	0,78	1	0,5	1	0,67
callcode	0,78	1	0,5	1	0,67

Table 6.5: Results by low-level constructs for Slither **function** contracts

SLITHER constructor	**Accuracy**	**Precision**	**Recall**	**Specificity**	**F_1-Score**
send	0,75	1	0,5	1	0,67
call	0,75	1	0,5	1	0,67
delegatecall	0,75	1	0,5	1	0,67
staticcall	0,75	1	0,5	1	0,67
callcode	0,75	1	0,5	1	0,67

Table 6.6: Results by low-level constructs for Slither **constructor** contracts

missed category of contracts, as described above. We see no difference in the detection ability among the low-level constructs. The results are presented in table 6.5 and 6.6.

ConFuzzius performs well for `send` and `call` constructs, but does not detect weaknesses in `delegatecall`, `staticcall` and `callcode` at all. The results are displayed in table 6.7.

In the accompanying paper [TIGS21][p.16], the authors of ConFuzzius explain how the detection of "Unhandled Exception" operates:

> "**Unhandled Exception.** We detect unhandled exceptions by first checking if the execution trace contains a CALL instruction that pushes to the stack the value 1 as a result of the call. A value of 1 means that an error occurred during the call (i.e. an exception). Afterwards, we check if the result of the call flows into a JUMPI instruction. If the result does not flow into a JUMPI

CONFUZZIUS function	**Accuracy**	**Precision**	**Recall**	**Specificity**	**F_1-Score**
send	0,89	0,8	1	0,8	0,89
call	0,89	0,8	1	0,8	0,89
delegatecall	0,56	-	0	1	-
staticcall	0,56	-	0	1	-
callcode	0,56	-	0	1	-

Table 6.7: Results by low-level constructs for ConFuzzius **function** contracts

MYTHRIL function	**Accuracy**	**Precision**	**Recall**	**Specificity**	**F_1-Score**
0.4.26	0,89	0,8	1	0,8	0,89
0.5.17	0,89	0,8	1	0,8	0,89
0.6.12	0,89	0,8	1	0,8	0,89
0.7.6	0,89	0,8	1	0,8	0,89
0.8.29	0,6	0,8	0,14	0,97	0,24

Table 6.8: Results by Solidity compiler version for Mythril for **function** contracts

> instruction until the end of the execution trace, then this means that the exception of the call was not handled and we report an unhandled exception."

However, when examining the code of the "unchecked_return_value" detector [1], it suggests handling all call-based low-level constructs.

6.1.3 Results by Solidity Compiler Version

Taking a different point of view, we split the results into different Solidity compiler versions.

The results for **Mythril**, table 6.8 and 6.9, show consistent performance from version 0.4.26 to 0.7.6 for function and constructor contracts with high accuracy, recall, and specificity, indicating reliable detection of both vulnerable and safe contracts. However, in version 8.9.29 there is a significant drop in accuracy and recall, which is also reflected in the low F_1-Score. This suggests that Mythril fails to detect most weaknesses in contracts compiled with version 0.8.29. We dedicate a separate section to these findings; see more in section 6.3.

For **Slither**, table 6.10 and 6.11, and **ConFuzzius**, table 6.12, the stable results indicate that the Solidity compiler version has no influence on the detection ability.

[1] `https://github.com/christoftorres/ConFuzzius/blob/master/fuzzer/detectors/unchecked_return_value.py`

MYTHRIL constructor	**Accuracy**	**Precision**	**Recall**	**Specificity**	**F_1-Score**
0.4.26	1	1	1	1	1
0.5.17	1	1	1	1	1
0.6.12	1	1	1	1	1
0.7.6	1	1	1	1	1
0.8.29	0,57	1	0,14	1	0,25

Table 6.9: Results by Solidity compiler version for Mythril for **constructor** contracts

SLITHER function	**Accuracy**	**Precision**	**Recall**	**Specificity**	**F_1-Score**
0.4.26	0,78	1	0,5	1	0,67
0.5.17	0,78	1	0,5	1	0,67
0.6.12	0,78	1	0,5	1	0,67
0.7.6	0,78	1	0,5	1	0,67
0.8.29	0,78	1	0,5	1	0,67

Table 6.10: Results by Solidity compiler version for Slither for **function** contracts

SLITHER constructor	**Accuracy**	**Precision**	**Recall**	**Specificity**	**F_1-Score**
0.4.26	0,75	1	0,5	1	0,67
0.5.17	0,75	1	0,5	1	0,67
0.6.12	0,75	1	0,5	1	0,67
0.7.6	0,75	1	0,5	1	0,67
0.8.29	0,75	1	0,5	1	0,67

Table 6.11: Results by Solidity compiler version for Slither for **constructor** contracts

CONFUZZIUS function	**Accuracy**	**Precision**	**Recall**	**Specificity**	**F_1-Score**
0.4.26	0,7	0,8	0,43	0,91	0,56
0.5.17	0,7	0,8	0,43	0,91	0,56
0.6.12	0,7	0,8	0,43	0,91	0,56
0.7.6	0,7	0,8	0,43	0,91	0,56
0.8.29	0,7	0,8	0,43	0,91	0,56

Table 6.12: Results by Solidity compiler version for ConFuzzius for **function** contracts

	Mythril f	Mythril c	Slither f	Slither c	ConFuzzius f
Mythril f	1	-	0,45	-	0,42
Mythril c	-	1	-	0,45	-
Slither f	0,45	-	1	-	0,3
Slither c	-	0,45	-	1	-
ConFuzzius f	0,42	-	0,3	-	1

Table 6.13: Results Jaccard Index

6.2 Similarity and Overlap

In addition to the metrics presented above, we are interested in the similarity of the result sets of the tools. We compare the sets of flagged contracts among the vulnerable contracts by each tool, split again into function and constructor contracts.

Since ConFuzzius cannot detect constructor contracts, we only compare it to Mythril's and Slither's flagged function contracts. The results of the calculation of the Jaccard similarity are presented in table 6.13.

The Jaccard similarity between Mythril and Slither is 45% for both function contracts and constructor contracts, indicating that a significant number of contracts are flagged by only one tool. The similarity is 42% between Mythril and ConFuzzius for function contracts and 30% between Slither and ConFuzzius, again for function contracts, showing a lower overlap between Slither and ConFuzzius compared to Mythril and ConFuzzius.

The overlap analysis reveals the following results:

- 83% of the flagged contracts reported by Slither were also reported by Mythril. In contrast, only 50% of the findings reported by Mythril are also identified by Slither. These numbers are, again, for both function and constructor contracts.
- 87% of the flagged function contracts reported by ConFuzzius were also reported by Mythril, whereas only 45% of the findings reported by Mythril are also identified by ConFuzzius.
- 50% of the flagged function contracts reported by ConFuzzius were also reported by Slither. In the other direction, 43% of the findings reported by Slither are also identified by ConFuzzius.

The computations regarding the Jaccard similarity and the overlap can be found in `compute_jaccard.py`.

Summarising the findings, we present in table 6.14 an overview of the detection abilities of the three tools based on our data set.

Mythril, Slither, ConFuzzius	0.4.26	0.5.17	0.6.12	0.7.6	0.8.29
send (function)	• • •	• • •	• • •	• • •	• • •
call (function)	• • •	• • •	• • •	• • •	◦ • •
delegatecall (function)	• • ◦	• • ◦	• • ◦	• • ◦	◦ • ◦
staticcall (function)	-	• • ◦	• • ◦	• • ◦	◦ • ◦
callcode (function)	• • ◦	-	-	-	-
send (constructor)	• • ◦	• • ◦	• • ◦	• • ◦	• • ◦
call (constructor)	• • ◦	• • ◦	• • ◦	• • ◦	◦ • ◦
delegatecall (constructor)	• • ◦	• • ◦	• • ◦	• • ◦	◦ • ◦
staticcall (constructor)	-	• • ◦	• • ◦	• • ◦	◦ • ◦
callcode (constructor)	• • ◦	-	-	-	-

Table 6.14: Overview of detection ability: A triple represents the detection ability of Mythril, Slither, ConFuzzius; • - can be detected, ◦ - cannot be detected

6.3 Mythril and Solidity 0.8.29

After the test runs with our generated data set, we attempted to contact the developers of Mythril by sharing our findings on the project's GitHub issue page [2], as well as on the community Discord server and asking for insight into the observed behaviour. However, as of the thesis deadline we had not received any responses within 3 months.

To further investigate possible changes in Mythril's behaviour, we started by executing the tool on the generated data set in bytecode mode, which expectedly revealed no differences in the results. To specifically analyse issues related to call-based low-level constructs, a handcrafted set of additional contracts was created, which is also available in the thesis' repository.

The focus is on comparing Solidity compiler versions 0.7.6 and 0.8.29 and the low-level constructs `call`, `delegatecall`, and `staticcall`, where the payload value is varied. This results in 60 additional contracts: 30 per Solidity version, 10 per low-level construct with 5 checked and 5 vulnerable contracts each.

The following observations were made: While for version 0.7.6 all contracts from the additional data set were correctly flagged, Mythril does not flag contracts with version 0.8.29 and

- `callAddress.call(abi.encodeWithSignature("foo()"));` nor
- `callAddress.call(abi.encodeWithSignature(""));`

as vulnerable, as well as the respective variants with `delegatecall` and `staticcall`.

[2] https://github.com/ConsenSysDiligence/mythril/issues/1908

Only contracts with `callAddress.call("");` are correctly classified as vulnerable.

Detailed results and contracts can be found in the thesis' repository.

CHAPTER 7

Conclusion

Blockchain technology and smart contracts are a rapidly growing field that offers new opportunities and new challenges, especially from a security perspective.
In this thesis, we investigated the weakness of unchecked low-level call return values and analysed how well selected detection tools can identify its different variants. Low-level language constructs, such as `send`, `call`, `delegatecall`, `staticcall`, and `callcode` do not throw an exception when confronted with an error, but return a boolean value which is false upon failure. This return value has to be checked by developers, for example, through the `require` statement, before continuing with computations. Missing this check can lead to unwanted program behaviour, such as program continuation under false assumptions.

There are numerous vulnerability detection tools available today. Some are developed primarily for scientific and research purposes, while others are widely adopted in practical smart contract development. These tools use a variety of detection approaches, each with its own strengths and limitations. One of the most popular techniques is symbolic execution, which systematically explores execution paths of a contract using symbolic inputs instead of concrete values, thus achieving high code coverage.

We developed a framework for the generation of minimal contracts that represent several variants of the weakness. In addition to that, we placed the weakness in the contracts' constructor as well as in functions and compiled them with different Solidity compiler versions from 0.4 to 0.8. With this data set, three vulnerability scanners were tested. Mythril and Slither are static analysis tools, and ConFuzzius is a dynamic analysis tool employing a hybrid fuzzing approach.

The results reveal the strengths and shortcomings of the tools.
Mythril has the most balanced detection performance overall, showing consistent results across different low-level constructs, most Solidity versions, and both constructor- and

function-based contracts. However, Mythril could not detect the weakness in call-based contracts with Solidity version 0.8.29.

Slither detects weaknesses in function and constructor code across all low-level constructs and Solidity compiler versions. It misses one category of contracts: the variant in which the return value of the low-level call is assigned to a variable, but not checked.

ConFuzzius only analyses runtime code and calls functions with fuzzed input; hence, it cannot detect weaknesses placed in the constructor. The Solidity compiler version has no influence on the detection ability. It performs well for `send` and `call`, but cannot detect weaknesses in `delegatecall`, `staticcall`, nor `callcode`.

Apart from the individual behaviour of each tool, we were interested in the similarity of the sets of flagged contracts. The analysis showed that in all comparisons, the similarity remains below 50%, which means that fewer than half of the contracts identified by one tool are also detected by the other. The overlap analysis, the unidirectional comparison, reveals strong asymmetries between the tools. One tool often confirms a large part of another's findings, while the reverse direction shows lower agreement.

The study showed valuable insights into the capabilities of different detection tools, highlighting their strengths and shortcomings, and it sharpened our senses for smart contract security.
For future work, our study could be extended in several directions. First, applying the same data set to a broader range of vulnerability detection tools would provide a more comprehensive comparison and allow us to evaluate how other tools handle this weakness. Second, the framework itself could be extended by further variants of the weakness to capture a wider spectrum of programming patterns and corner cases. Such extensions would contribute to a deeper understanding of smart contract weaknesses and the strengths and limitations of current smart contract analysis approaches.

Overview of Generative AI Tools Used

ChatGPT was used to improve sentence formulation and Overleaf's Writefull integration for spelling and wording suggestions.

List of Figures

List of Tables

Bibliography

[ADFS23] Monika Di Angelo, Thomas Durieux, João F. Ferreira, and Gernot Salzer. SmartBugs 2.0: An Execution Framework for Weakness Detection in Ethereum Smart Contracts. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 2102–2105. IEEE, 2023.

[AMS24] Monika Di Angelo, Rafael Mohr, and Gernot Salzer. Systematic Study of Compilers and Vulnerability Scanners Using the Example of Integer Bugs. In *IEEE International Conference on Blockchain, Blockchain 2024, Copenhagen, Denmark, August 19-22, 2024*, pages 243–251. IEEE, 2024.

[BCD+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Comput. Surv.*, 51(3), 2018.

[But14] Vitalik Buterin. Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform. `https://ethereum.org/en/whitepaper/`, 2014.

[cwea] Common Weakness Enumeration - Vulnerability. `https://cwe.mitre.org/documents/glossary/index.html#Vulnerability`. Accessed: July 2025.

[cweb] Common Weakness Enumeration - Weakness. `https://cwe.mitre.org/documents/glossary/index.html#Weakness`. Accessed: July 2025.

[dADFS23] Monika di Angelo, Thomas Durieux, João F. Ferreira, and Gernot Salzer. Evolution of Automated Weakness Detection in Ethereum Bytecode: a Comprehensive Study, 2023.

[DFAC20] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. Empirical Review of automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 530–541. ACM, June 2020.

[dFC21] Luciano da F. Costa. Further Generalizations of the Jaccard Index. *CoRR*, abs/2110.09619, 2021.

[evm25] Ethereum Virtual Machine (EVM). `https://ethereum.org/en/developers/docs/evm/`, 2025. Accessed: July 2025.

[FGG19] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2019, Montreal, QC, Canada, May 27, 2019*, pages 8–15. IEEE / ACM, 2019.

[Fü25] Lisa Fürst. Reentrancy in Ethereum Smart Contracts: Methods and Tools for Detection. Diploma thesis, Technische Universität Wien, 2025.

[Gor21] Philippos Gorgoris. Identifying administrators of smart contracts from transaction data. Diploma thesis, Technische Universität Wien, 2021.

[int25] Introduction to Smart Contracts. `https://docs.soliditylang.org/en/latest/introduction-to-smart-contracts.html`, 2025. Accessed: July 2025.

[KYCO23] Chihiro Kado, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. An Empirical Study of Impact of Solidity Compiler Updates on Vulnerabilities. In *IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events, PerCom Workshops 2023, Atlanta, GA, USA, March 13-17, 2023*, pages 92–97. IEEE, 2023.

[MKM+24] Charalambos Mitropoulos, Maria Kechagia, Chrysostomos Maschas, Sotirios Ioannidis, Federica Sarro, and Dimitrios Mitropoulos. Broken Agreement: The Evolution of Solidity Error Handling. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2024, Barcelona, Spain, October 24-25, 2024*, pages 257–268. ACM, 2024.

[Mue18] Bernhard Mueller. Smashing Ethereum Smart Contracts for Fun and Real Profit. *HITB SECCONF Amsterdam*, 9(54):4–17, 2018.

[owa25] OWASP Smart Contract Top 10. `https://owasp.org/www-project-smart-contract-top-10/`, 2025. Accessed: July 2025.

[RAS22] Heidelinde Rameder, Monika Di Angelo, and Gernot Salzer. Review of Automated Vulnerability Analysis of Smart Contracts on Ethereum. *Frontiers Blockchain*, 5, 2022.

[sol25a] Solidity Documentation - Expressions and Control Structures. https://docs.soliditylang.org/en/v0.8.29/control-structures.html, 2025. Accessed: July 2025.

[sol25b] Solidity Documentation - Type Address. https://docs.soliditylang.org/en/v0.8.29/types.html#address, 2025. Accessed: July 2025.

[swc] Smart Contract Weakness Classification Registry - 104. https://swcregistry.io/docs/SWC-104/. Accessed: July 2025.

[swc18] Smart Contract Weakness Classification Registry. https://swcregistry.io/, 2018. Accessed: July 2025.

[Sza94] Smart Contracts. https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html, 1994. Accessed: July 2025.

[Tha21] Alaa Tharwat. Classification Assessment Methods. *Applied Computing and Informatics*, 17(1):168–192, 2021.

[TIGS21] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*, pages 103–119. IEEE, 2021.

[VIL24a] Fernando Richter Vidal, Naghmeh Ivaki, and Nuno Laranjeiro. OpenSCV: an open hierarchical taxonomy for smart contract vulnerabilities. *Empir. Softw. Eng.*, 29(4):101, 2024.

[VIL24b] Fernando Richter Vidal, Naghmeh Ivaki, and Nuno Laranjeiro. Vulnerability detection techniques for Smart Contracts: A systematic literature review. *J. Syst. Softw.*, 217:112160, 2024.

[VR22] Darin Verheijke and Henrique Rocha. An exploratory study on Solidity guards and Ether exchange constructs. In *Proceedings of the 5th International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2022, Pittsburgh, Pennsylvania, 19 May 2022*, pages 1–8. ACM, 2022.

[WSC+18] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. Towards optimal concolic testing. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 291–302. ACM, 2018.

[YMRS19] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. Blockchain Technology Overview. *CoRR*, abs/1906.11078, 2019.