Towards the Assessment and the Improvement of Smart Contract Security

Settore Scientifico Disciplinare INF/01

Dottorando:
Mirko Staderini

Tutore
Prof. Andrea Bondavalli

Coordinatore
Prof. Matteo Focardi

Anni 2018/2021

ABSTRACT

Blockchain technologies (hereafter called Blockchain) allow storing information guaranteeing properties such as immutability, integrity and non-repudiation of data. Although Blockchain is not a panacea, this technology has rapidly evolved in recent years. The development of smart contracts (which automatically execute computerized transactions) has increased the application areas of the Blockchain. One of the most important issues is security; the problem is even more critical, considering that smart contracts cannot be patched once they are deployed into the Blockchain.

Ethereum is one of the main platforms for smart contract development, and it offers Solidity as its primary (and Turing-complete) language. Solidity is a new language which evolves rapidly. As a result, vulnerability records are still sparse, and consequently, the existing smart contract checking tools are still immature. On the other hand, Solidity is just another new programming language reusing its central notions from traditional languages extended by Ethereum-specific elements. Then, the most promising way to create a quality assurance process is adapting more general existing technologies to the peculiarities of Ethereum and, in particular, Solidity.

Unfortunately, despite various studies and trials on the subject, no literature approach clearly solves the problems related to the vulnerability of smart contracts. To contribute to this hot field, we propose our methodology to assess and improve the smart contract security. At first, we address the problem of overcoming the Solidity rapid evolution through the definition of a set of 32 vulnerabilities and their language-independent classification in 10 categories. Then, we assess smart contract security by applying one of the most popular approaches to discover vulnerabilities: *static analysis* (SA). After selecting static analysis tools, we identify categories of vulnerabilities that SA tools cannot cover.

Next step is to conduct an experimental campaign based on the analysis of contracts across the selected toolset. We realized that processing smart contracts, randomly extracted from Etherscan (a Blockchain explorer) with SA tools results in several positives. We determined thus, overall and for each category of vulnerabilities, the best-built tools (wrt. their effectiveness against the subset of

vulnerabilities they target) and the most effective ones (wrt. the entire vulnerability set).

We found a lack of coverage of vulnerabilities in using each and every tool individually. This lack took us to the investigation of possible approaches to improve the security of smart contracts. A first approach has been to use several tools in a combined way to increase the coverage. Through this analysis we determined also the combinations with the highest coverage. Then we analyzed those vulnerabilities that escape the detection so to provide an ordering for deciding which vulnerabilities should be addressed first in the process of modifying static analysis tools to improve their coverage. As a last contribution, we investigated how to improve the tool effectiveness by determining where vulnerabilities are most likely located.

ACKNOWLEDGEMENTS

We all faced difficult times during the last years. Covid and complex personal situations have marked this period almost immediately, making it more complicated than it should have been. Often we were all forced to stay at home, causing the lack of that dynamism, that exchange of ideas that only interpersonal contact, the exchange with different cultures, can generate.

In this context, I want to thank my professor Andrea Bondavalli, for being the best guide I could have asked for. Regardless of his schedule, he always carved out time to answer my questions and concerns. He has been invaluable with his constant prodding and advice. In fact, he has been much more than that, being supportive on a personal level as well.

I would like to thank the entire RCL group (I want to mention in particular Paolo, Tommaso, Andrea), who tried to support me in every way. I felt welcomed inside a family, something not at all taken for granted and really precious.

I sincerely thank Prof. András Pataricza, who gave me food for thought and helped me in many moments of this journey.

I thank all the people I worked with during my PhD (Imre, Akos, Oskár, Balasz, Nuno, to name a few).

In addition to the people associated with the university, the most important thanks go to my wife. She has put up with me, supported me beyond belief. She straightened the tiller of the rudder even when it was crooked. To my parents. To my sister and nieces. To my colleagues and superiors without whose support I could not have been away for so long.

Let me also give a thought, a big thought to those who are no longer here. Giancarlo, Marina, Grandma Anna, Grandpa Ernesto. Their love is forever etched in my heart.

LIST OF PUBLICATIONS

The following publications have been produced in the context of the research work described in this Thesis.

SRDS 2018 Staderini Mirko, Schiavone Enrico, and Bondavalli Andrea. “A Requirements-Driven Methodology for the Proper Selection and Configuration of Blockchains,” in *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, Oct. 2018, vol. 2019, pp. 201–206, doi: 10.1109/SRDS.2018.00031.

MINISY 2020 Staderini Mirko, and Palli Caterina. "An Analysis on Ethereum Vulnerabilities and Further Steps," *27th Minisymposium of the Department of Measurement and Information Systems*. Budapest University of Technology and Economics, 2020.

BCCA 2020 Staderini Mirko, Palli Caterina, and Bondavalli Andrea. “Classification of Ethereum Vulnerabilities and their Propagations,” in *2020 Second International Conference on Blockchain Computing and Applications (BCCA)*, Nov. 2020, pp. 44–51, doi: 10.1109/BCCA50787.2020.9274458.

MINISY 2021 Staderini Mirko, and Bondavalli Andrea. " Investigating Static Analyzers Detection Capabilities on Ethereum Smart Contracts," *28th Minisymposium of the Department of Measurement and Information Systems*. Budapest University of Technology and Economics, 2021.

BLOCKCHAIN ’21 Staderini Mirko, and Bondavalli Andrea. “Investigation on Vulnerabilities Location in Solidity Smart Contracts”. In: Prieto J., Partida A., Leitão P., Pinto A. (eds) Blockchain and Applications. BLOCKCHAIN 2021. Lecture Notes in Networks and Systems, vol. 320. Springer, Cham, pp. 199-211, doi: 10.1007/978-3-030-86162-9_20.

LADC 2021 Staderini Mirko, Pataricza Andràs, and Bondavalli Andrea. “Static Analysis Tools Applied to Smart Contracts,” X Latin-American Symposium on Dependable Computing (LADC 2021) - Fast Abstract, doi: 10.5753/ladc.2021.18529.

Moreover, the following work contains submitted but not yet revised material, and therefore is reported separately.

Staderini Mirko, Pataricza Andràs, and Bondavalli Andrea. “Security Evaluation and Improvement of Solidity Smart Contracts,” submitted to January 2022.

.

This page is intentionally left blank

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

1.1 Preliminary and Motivations

Blockchain technologies [1] (hereafter referred simply as Blockchain) are characterized by a shared database (or ledger) distributed across a peer-to-peer network. The term recalls the structure, a chained sequence of blocks, where each block contains a set of transactions and, except for the first one called genesis block, it is linked to its predecessor by means of a cryptographic hash. Blockchain, while not a panacea [2], promises an out-of-the-box solution to improve the security of distributed systems. Smart contracts are one of the most important innovations of the second generation of the Blockchain. The basic idea is to execute computerized transactions automatically. Their diffusion has allowed the development of applications in different areas (e.g., financial, medical, insurance, gaming, betting). Blockchain protects smart contracts, data, and transaction logs by a strong hash encoding, thus ensuring their immutability and non-repudiability.

However, design and coding faults and *weaknesses* in the smart contracts implementing the particular application can still result in exploitable *vulnerabilities* to malicious attacks despite the well-designed run-time environment. This problem is even more critical, considering that developers cannot patch smart contracts once deployed on the Blockchain. Thus, it is crucial to identify security flaws in the code at the early stage of the development life cycle. Vulnerabilities in smart contracts can lead to severe consequences: an example is the financial losses caused by the DAO attack [3] that allowed the attacker to steal around US$60M worth of cryptocurrency.

Ethereum [4] is one of the most widely used platforms for smart contracts and Solidity [5] is the primary programming language that targets the development of Ethereum smart contracts. One of the major open problems related to the Ethereum blockchain is the insufficient quality assurance. Vulnerable records are still sparse due to the novelty and rapid evolution of Solidity technology. Consequently, the existing smart contract checking tools are still immature. On the other hand, Solidity is just another new programming language reusing central notions from traditional ones extended by Ethereum-specific elements. Therefore, the most promising way to create a quality assurance process for Solidity is adapting existing technologies to the peculiarities of Ethereum and, in particular, Solidity.

1.2 Our Contribution

In the literature there are no clear answers on how to provide guarantees for quality assurance for smart contracts. To contribute to filling this gap, in this thesis we propose and discuss our approach towards the assessment and the improvement of the security of smart contracts. We deal only with Solidity Versions 0.5 and upper (the latest release is 0.8) due to the incompatibility with previous versions.

New vulnerabilities emerge as the language evolves; analyses too tied to a specific language release quickly become obsolete. The main reason is the novelty and the rapid evolution of Solidity. Considering previous studies, we noticed a lack of agreement in the identification of the number of vulnerabilities and their systematization. The missing agreement leads to user confusion and vulnerabilities proliferation, as well as a difficulty for researchers to compare weaknesses on different platforms. Moreover, existing classifications either do not abstract from a specific Solidity release or do not capture the behaviour of vulnerabilities; thus, they are too dependent on the language release.

The first research question we address is: *How to overcome the language evolution?* (**RQ1**).

Software security engineering has solid empirical foundations from a well-organized and maintained process of vulnerability data acquisition, abstraction, and generalization. The Common Vulnerabilities and Exposures (CVE) database collects, defines, and catalogues publicly disclosed cybersecurity vulnerabilities, i.e., weaknesses in software (and hardware) components that, when exploited, spoil the security of the system. The Common Weakness Enumeration (CWE) classifies weaknesses as root causes of vulnerabilities into a hierarchical taxonomy; furthermore, each CWE list item highlights the mode of introduction, expected consequences, and potential mitigations. The highly abstract top two categories of weaknesses in the hierarchy are already independent of any specific language or technology.

We aim to provide a Solidity-specific vulnerability analysis, categorizing each vulnerability with classes based on a general-purpose (not version or language-specific) classification. Moving beyond language evolution is the foundation for the next steps in our research. As the CWE has a primary security focus, we systematize vulnerabilities and provide a Solidity fault model based on CWE (comparing it with the ISO 5055:2021 standard [6]).

Static analysis (SA) is one of the most significant and widely used types of code analysis. It inspects the code without executing it. At first, it extracts an abstract model of the code under evaluation. It searches for potential vulnerabilities in the code over the model by looking for weaknesses (antipatterns). Its low effort demand compensates for its incomplete (but for most applications still sufficient) detection coverage [7].

The second main research question we address is: *How can we evaluate the security of smart contracts by using static analysis to detect the most relevant vulnerability-related weaknesses?* (**RQ2**).

The research into applying static analysis to detect vulnerabilities and weaknesses in Ethereum smart contracts increased significantly after the first infamous exploits in 2016 [3]. Several static analyzers have been developed in the last years, focusing explicitly on vulnerability detection of Solidity smart contracts. Several works compared static analyzers applied to smart contracts.

To tackle this problem, we use the Solidity vulnerability model as a basis. Performing an analysis of the capabilities of some selected SA tools to detect weakness originating vulnerabilities on a representative set of smart contracts permits to:

- evaluate the individual tool behaviour on its anticipated set of vulnerabilities-related weaknesses. This way, we can identify vulnerability classes escaping detection by the particular SA tool;
- assess the smart contract security by computing basic statistical metrics for comparing the detection capabilities of different SA tools;
- These assessments permit to: i) quantify tools anticipated vulnerability model and related testing quality; ii) build a benchmark of the tools when exposed to a generic set of smart contracts.

Using static analysis permits to assess the security of smart contracts. However, single tools have highly different classwise detection capabilities, and several vulnerabilities escape the detection.

The third research question is: *How to improve the smart contract security using SA tools?* (**RQ3**).

A first promising way to improve smart contract security is to combine several tools for coverage improvement at the price to increase the number of false positives.

Even using combinations of tools, we have undetected vulnerabilities (false negatives). False negatives are dangerous because they instil unfounded confidence in the code correctness. A question arises. Are all FNs equally relevant? Identifying the most critical types of undetected vulnerabilities allows defining the top priority in planning an effort for mitigation. We want to investigate whether they are all equally critical or if some can be more dangerous, thus allowing to improve security by giving priority to the most critical vulnerabilities.

Vulnerabilities and analysis tools are a widely debated topic. However, the characterization of the position of vulnerabilities in Solidity smart contracts is surprisingly less investigated compared to other programming languages. A third way for improvement of analysis and understanding is finding where a specific class of vulnerabilities is located into smart contracts. On one side, tool developers can be guided to improve the vulnerability detection capabilities of the tool. On the other side, software developers can produce more secure contracts focusing on the specific areas where such vulnerabilities are more likely located.

1.3 THESIS STRUCTURE

The remainder of the Thesis proceeds as follows.

Chapter 2 introduces basic concepts related to dependability, Blockchain, smart contracts and static analysis. Other chapters are built on these bases.

Chapter 3 investigates the first problem we set out to address. After determining the available collections of Solidity vulnerabilities, it provides a model for Solidity and then determines vulnerability propagations (RQ1). We use this model as a basis for the following chapters.

Chapter 4 focuses on assessing smart contract security using static analysis tools. After identifying a set of static analysis tools, by analysing the tools we determine for each of them the targeted set of weaknesses to detect. Next, an intensive experimental campaign permits determining how tools behave on their targeted detectable vulnerabilities as well as in dealing with the entire set of vulnerabilities identified. (RQ2).

Chapter 5 addresses in different ways how to improve the security of smart contracts. First, an analysis of the detection capabilities of tool combination is performed; then a study targets the criticality of the uncovered vulnerabilities to define priorities of further actions to treat them. Finally, it investigates where vulnerabilities are more likely located (RQ3).

Chapter 6 concludes the Thesis. An archive of support files is available at [124].

2 BASIC CONCEPTS AND RELATED WORKS

This chapter describes the context in which this thesis was developed. To do so, Section 2.1 reports on the dependability, security and related concepts, Sections 2.2 and 2.3 respectively discuss the basics of Blockchain and smart contracts, and Section 2.4 introduces static analysis. The state-of-the-art and related work are introduced within the description below and detailed in Section 2.5.

2.1 Dependability and Security

2.1.1 *Basic Definitions*

According to [8], a **system** is *an entity that interacts with other entities (*i.e., other systems - the **environment** of the given system - including hardware, software, humans and the physical world). A **function** of a system is *what the system is intended to do*, when its **behaviour** is *what the system does to implement its function*, and it is described as a sequence of states. The **global state** of a system is the set of states related to computation, communication, stored information, interconnection and physical conditions.

A **service** delivered by a system (the provider) *is the behaviour as it is perceived by its* **user** (another system that receives the service**).** The service is the sequence of the (external) states of the provider. A service **interface** is the *part of the system boundary of the provider where service delivery takes place*. An **external state** is the *part of the total state of the provider perceivable at the interface;* the **internal state** is the remaining part of the total state.

The **life cycle** of a system encompasses, among others, two phases: the **development** and **use** phase. The *development* phase includes all activities until the system is ready to deliver service. The *use* phase starts when the system begins to deliver its service to the user.

According to [9], an **Autonomous System** is a system that *can provide its services without guidance by another system*. By considering a **Subsystem as** a system that is a part of an encompassing bigger system, a **Constituent System** (CS) is *an autonomous subsystem of an SoS, consisting of computer systems and possibly of controlled objects and/or human role players that interact to provide a given service*.

A **System-of-Systems** (SoS) is an integration of a finite number of CSs which are independent and operable, and which are networked together for a period of time to achieve a certain higher goal. A **Cyber-Physical System** (CPS) is a system consisting of a computer system (the cybersystem), a controlled object (a physical system) and possibly of interacting humans.

The original definition [8] of **dependability** stresses the need for justification of trust: *Dependability is the ability to deliver service that can justifiably be trusted*.

An alternative definition focuses on services and considers the dependability *as the ability to avoid service failures that are more frequent and more severe than is acceptable*.

2.1.2 *Dependability Attributes and Security*

Dependability is an integrating concept that includes the following attributes [8]:

- *Availability*: readiness for correct service.
- *Reliability*: continuity of correct service.
- *Safety*: absence of catastrophic consequences on the user and the environment.
- *Integrity*: absence of improper system alterations.
- *Maintainability*: ability to undergo modifications and repairs.

Defining another attribute (confidentiality) permits the introduction of the concept of security.

Confidentiality: absence of unauthorized disclosure of information.

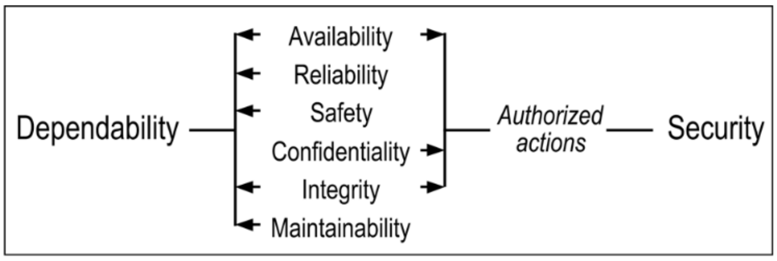


Figure 1: Relationship between dependability and security.

Security. *The composition of confidentiality, integrity, and availability; security requires the concurrent existence of availability for authorized actions only, confidentiality, and integrity with "improper" meaning "unauthorized".*

The relation between security and dependability attributes are highlighted in Figure 1 (from [8]). The dependability and security specification of a system must include the requirements for the attributes in terms of the acceptable frequency and severity of service failures for specified classes of faults and a given user environment. One or more attributes may not be required at all for a given system.

2.1.3 *Threats: Failures, Errors, Faults*

Following [8], when the service implements the function of the system, a **correct service** is delivered. A **service failure** (**failure**) occurs when the delivered service deviates from the correct service. An **error** is a deviation between the system state and the correct service state. The adjudged or hypothesized cause of an error is called a **fault**.

A fault can be internal or external from the system. Faults can be classified into eight basic viewpoints: *phase of creation or occurrence, system boundaries, phenomenological cause, dimension, objective, intent, capability, persistence*. Combining the classes of faults allows identifying the most likely 31. Figure 2 (from [8]) shows how the combined fault classes (on the left) belong to three major overlapping groupings (at the bottom):

- *Development faults* include all fault classes during the development;
- *Physical faults* include all faults that affect hardware;

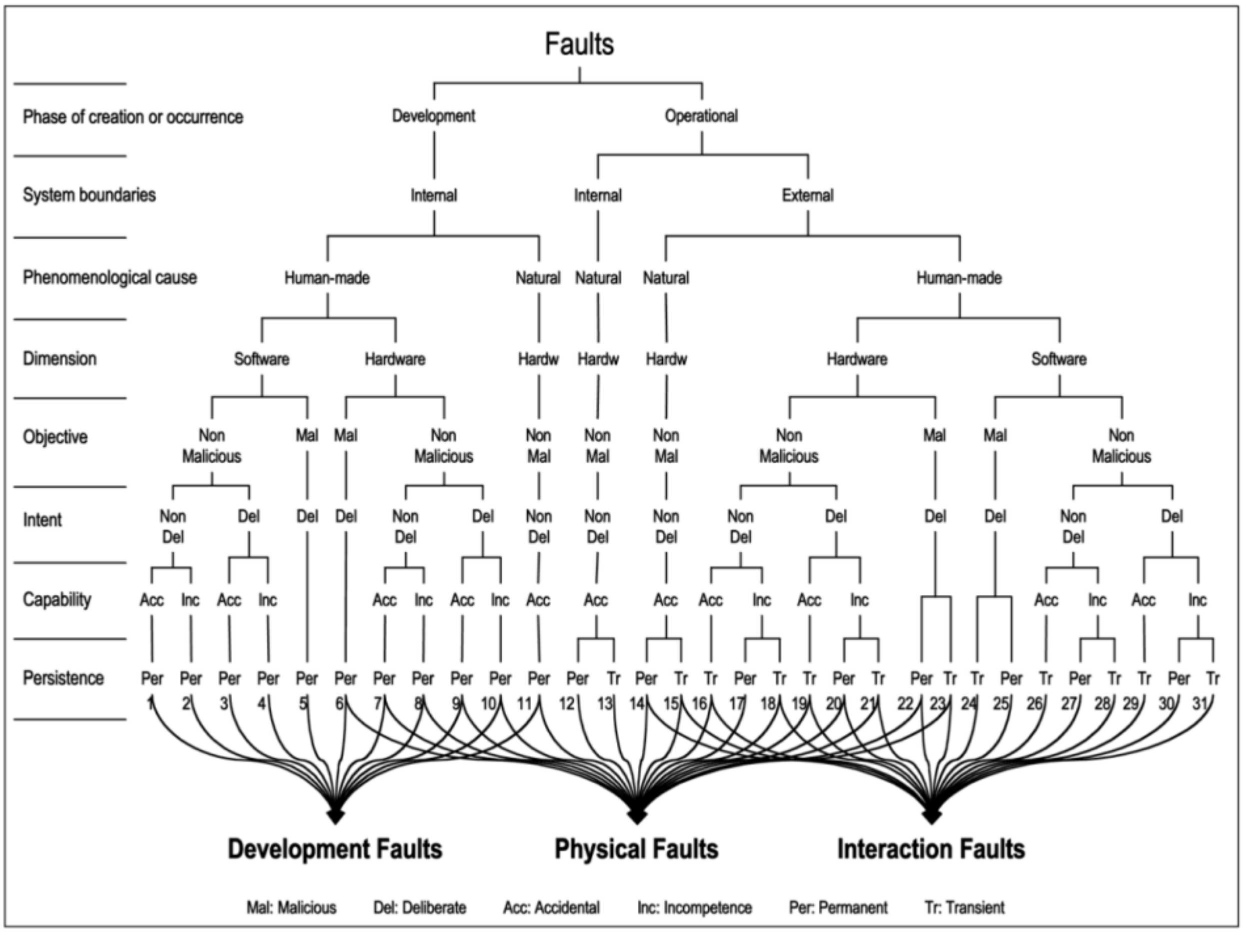


Figure 2: Classes of combined faults.

$$\begin{tikzcd}
\cdots \arrow[r] & \text{fault} \arrow[r, "\textit{activation}"] & \text{error} \arrow[r, "\textit{propagation}"] & \text{failure} \arrow[r, "\textit{causation}"] & \text{fault} \arrow[r] & \cdots
\end{tikzcd}$$

Figure 3: The chain of dependability and security threats.

- *Interaction faults* include all external faults.

Threats *can be summarized as failures, errors or faults*. Figure 3 identifies the chain of threats; arrows identify a causality relationship. A fault is active when it produces an error; otherwise, it is dormant. A failure occurs when an error propagates to the service interface. A failure of a system causes a permanent or transient external fault for the other system(s) that receive service from the given system.

From a security point of view, a **vulnerability** is *an internal fault that enables an external and malicious fault* (**attack**) to harm the system.

In other terms, according to [10], a **vulnerability** is a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy. An **attack** is defined as *an assault on system security that derives from an intelligent threat*; that is, an intelligent act that is a deliberate attempt to evade security services and violate the security policy of a system.

We conclude this section with the definition of **exploit**. An **exploit** is, in essence, a *software script that will exercise a system vulnerability; invoking the exploit is an operational, external, human-made, software, malicious interaction fault*. The vulnerability that an exploit takes advantage of is typically a software flaw that could be characterized as a developmental, internal, human-made, software, non-malicious, nondeliberate, permanent fault.

2.1.4 *Attaining Dependability*

Means to attain security and dependability can be grouped in four categories [8]:

- **Fault prevention:** techniques to prevent the occurrence or introduction of faults. Some examples are design review, testing and software engineering.

- **Fault tolerance:** techniques to avoid service failures in the presence of faults. Fault tolerance is achieved through error detection and recovery process.
- **Fault removal:** techniques to reduce the number and severity of faults. It consists in three steps: verification (checking whether the system adheres to given properties), diagnosis and correction.
- **Fault forecasting:** techniques to estimate the present number, the future incidence, and the likely consequences of faults.

The different emphasis on the various attributes influences the use of the means to make a system secure and dependable.

2.2 Blockchain Technologies (Blockchain)

In recent years, the interest in blockchain technologies (hereafter referred to as Blockchain) has grown exponentially. The reason for this excitement is ascribable to the ability to enable new forms of transactions and interactions between mistrusting and decentralized entities. Indeed, it has attracted interests and huge investments from enterprises; however, it is not a panacea and may even become useless or not convenient.

This section deals with introducing the basic Blockchain technological aspects ([1], [2], [11], [12]) constituting the background required in the following of the Thesis.

2.2.1 *Basics*

The Blockchain is a technology characterized by a shared database (or ledger) distributed across a peer-to-peer network. The term recalls its structure, a chained sequence of blocks, where each block contains a set of transactions and, except for the first one called *genesis* block, it is linked to its predecessor by means of a cryptographic hash. Blocks are linearly and chronologically added to the chain and can be seen as links of a constantly growing chain, hence the name *Blockchain*.

Each node of the network possesses a local replica of the Blockchain, which is updated every time after appending a block to the chain. The process of committing a block takes the name of *mining*, and the nodes which are taking care of validating transactions, collecting them into blocks and appending the blocks on the ledger are called miners. Nodes are typically independent peers capable of reaching an agreement on the status of the Blockchain, that is, the latest block to be appended, without the involvement of any central authority. This agreement is called *consensus*, and there are many different algorithms designed for reaching it (described in more detail in the dedicated Section 2.2.3).

The first application of Blockchain has been financial transactions of cryptocurrencies (known as Blockchain 1.0 [12]), and Bitcoin [1] is the most widespread and famous implementation. More recent alternatives enable systems and applications to record other kinds of information on the ledger. One example is distributed applications or smart contracts, executed and shared among participating entities, in which case the transaction includes the result of a function call. The smart contracts can be self-executing, and for a general-purpose, thanks to the Turing-completeness property provided in some cases, as for Ethereum [4]. These fundamental extensions of capabilities brought to the so-called second generation of blockchains, or Blockchain 2.0.

2.2.1.1 Fundamental Properties

Hereafter, we describe the fundamental properties typically provided in every distributed ledger; however, their provision may be only partial for some categories of Blockchain.

- *Immutability*. Due to the presence of cryptographic hashes in the blocks, transactions stored in the distributed ledger cannot subsequently be tampered with, reversed, or deleted without altering the hash values, thus without being detected.

- *Integrity*. Cryptography, together with algorithmic constraints, provides integrity on messages from users or between nodes and ensures that only authorized entities perform operations. In fact, Public Key Infrastructure (PKI) and digital signatures provide accounts identification and transactions authorization.

- *Non-repudiation*. It is the ability to protect against denial of an action (for example, having originated a transaction). In the context of the Blockchain, the sender digitally signs a transaction: in this way, the origin of each transaction is traced so that there is no dispute about it nor about their sequence in a distributed ledger. This guarantees, for example, the responsibility for monetary expenditure and the execution of smart contracts.

Table 1: Blockchain categories.

	Permissionless	**Permissioned**
Public	Reading is open Writing is open	Reading is open Writing is restricted
Private/Consortium	*Not used in practice*	Reading is restricted Writing is restricted

- *Transparency*. Each participating entity has access to the distributed ledger and can verify transactions without a central intermediary.
- *Decentralization.* There is no central authority deciding on recording a particular datum in the ledger. Also, decentralization avoids single points of failure.
- *Pseudo-anonymity.* In general, each user can interact with the Blockchain with a generated address. The address is a pseudonym that does not reveal the real identity of the user.

2.2.1.2 Blockchain categories

In some cases, the participants in the network may have different authorizations and play different roles [11], [13]. Thus, the decentralization obtained can be only partial. Let us consider the two main operations applicable to every database: reading and writing. In a Blockchain, *reading* consists in consulting the current state of the ledger and creating transactions. Instead, *writing* means validating transactions, aggregating transactions in blocks, appending blocks to the chain, and participating in the consensus protocol.

A *permissionless* blockchain is a decentralized and open system in which every node has reading and writing abilities. Examples of permissionless blockchains include Bitcoin [1] and Ethereum [4]. *Permissioned* blockchains, instead, have been proposed as an alternative in which only a set of known and identifiable participants, previously enrolled and admitted to the Blockchain, are allowed to read, write or perform both operations. Some state-of-the-art permissioned blockchains available today are Hyperledger Fabric, Ripple, Multichain, Kadena, Tendermint, and Chain.

The distinction between these classes is often combined with the notion of *public* and *private/consortium* blockchain, used to refer to reading permissions. The four resulting categories of Blockchain are shown in Table 1. Between them, public permissionless blockchains are diffused in businesses and applications involving the general population, as well as public permissioned ones but with the attribution of writing rights only to some privileged nodes. Private permissioned blockchains are typically owned by one institution (or a combination of institutions, which can be referred to as consortium blockchain). Private permissionless blockchains are not used in practice. They would restrict reading while permitting writing to any node; however, some applications may exist, e.g., a shared black box for air traffic, where writing is open to every aircraft and reading is read restricted to officers. Finally, choosing between permissionless and permissioned blockchains is not trivial, as there are often trade-offs including scalability, interoperability, cost, performance, availability, anonymity, privacy, confidentiality, transparency, and censorship resistance [2].

2.2.2 *Consensus Algorithms*

The way nodes reach an agreement on the status of the ledger is one of the most important components of a Blockchain: it affects performance (as transactions throughput and latency), as well as security and scalability. Many alternatives already exist, each with its own advantages and disadvantages. Table 2 gives a comparison between the mechanisms presented in the remainder of the section, which are the most widely used state-of-the-art consensus mechanisms ([14], [15], [16], [17], [18], [19]) highlighting their analogies and differences. The comparison is performed in terms of permission (permissioned – *p.ned*, permissionless – *p.less*), transaction finality (a transaction included in a block can be immediate – *det.*- or probabilistically -*prob.*- considered final), energy consumption, transaction rate, cost of participation, trust model and adversary tolerance (the percentage of the participants that can be malicious without affecting the algorithm).

Proof of Work (PoW). Every block contains a field named header, composed of metadata including, but not limited to, a timestamp and the hash of the previous block. Each miner node has to compute the header hash of the block to be appended. Solving this problem is not trivial: the block header is constantly changing, and the value must be equal or smaller than a given value. However, when a miner produces the PoW, all other nodes can easily verify the correctness of the value.

Table 2: A comparison of popular Blockchain consensus mechanisms.

	PoW	**PoS**	**PoET**	**BFT and variants**	**Ripple**	**SCP**
Permission	P.less	P.less	P.ned P.less	P.ned P.less	P.ned P.less	P.less
Transaction finality	Prob.	Prob.	Prob.	Det.	Det.	Det.
Energy Saving	No	Partial	Unknown	Yes	Yes	Yes
Transaction rate	Low	High	Medium	High	High	High
Token needed?	Yes	Yes	No	No	No	No
Cost of participation	Yes	Yes	No	No	No	No
Trust model	Untrusted	Untrusted	Untrusted	Semi-trusted	Semi-trusted	Semi-trusted
Adversary Tolerance	< 25% of computing power	< 51% of stake	Unknown	< 33.3%	<20% faulty nodes in Ripple's UNL	< 33%
Examples	Bitcoin, Ethereum	Cardano, Peercoin	Hyperledger	Hyperledger Fabric, NEO	Ripple	Stellar

After that, transactions in the proposed block are validated by peers to avoid fraud. If confirmed, the new block is added to the Blockchain. This is a real competition since the calculation is time- and energy-consuming. Thus, a reward is given to the winning miner. This algorithm is used in Bitcoin and Ethereum blockchains; block interval depends on different parameter settings (e.g., in Bitcoin, a block is generated about every 10 minutes [14], while in Ethereum 1.0 between 12 and 14 seconds [20]).

Proof of Stake (PoS). This algorithm requires the mining node to prove the ownership of some amount of cryptocurrency. The selection is based on stake size combined with many solutions (e.g., a formula favouring the lowest hash values or a random factor). PoS saves more energy and is more effective, while latency is shorter than PoW. However, the mining cost is close to zero, and it may attract attackers (nothing-at-stake) [18]. Ethereum is at phase 0 of the technical roadmap[1] for switching from PoW to PoS algorithm: the new solution promises to improve the transaction speed [21], and it offers protection against the Sybil attack. An efficient and power-saving variant of PoS is the *Delegated Proof of Stake* (DPoS), where an account may delegate its stake to others rather than validating transactions directly [11].

Proof of Elapsed Time (PoET). This algorithm was developed by Intel and used in the Hyperledger platform. It is based on leader election in a Trusted Execution Environment (TEE). The idea is to randomly assign an elapsed time to every node in the TEE, and elect the one that wins this lottery. This node has to prove that the time obtained is the lower, wait, and validate the block. Every other node in the TEE can easily perform the verification.

Byzantine Fault Tolerance (BFT) *and variants.* The so-called Practical Byzantine Fault Tolerance (PBFT) algorithm is the first solution to achieve the consensus in the presence of Byzantine failures, thus despite arbitrary behaviour from some nodes. It uses the concept of replicated state machines and voting by replicas for state changes. This algorithm requires "3f+1" replicas to tolerate "f" failing nodes. This approach imposes a low overhead on the performance of the replicated service. BFT-based Blockchain offers a much stronger consistency guarantee, lower latency, higher throughput, and it requires that all participants agree. Several variants and optimizations exist [22], [23]

[1] Phase 0 refers to December 2, 2021.

(e.g., XFT, parallel BFT, Hybrid BFT, Hierarchical BFT, Scalable BFT). BFT algorithms are recently used also in permissionless blockchains [23].

Ripple is based on the notion of the Unique Node List (UNL). Every server *s* considers only votes of its own UNL to determine consensus. Moreover, UNL represents a subset of network nodes in which *s* can trust collectively, but it cannot trust individually [24]. The algorithm is divided into rounds, and each round consists of four steps: 1) each server collects and inserts all the transactions in a set of candidates; 2) each server joins the sets of its own UNL and votes about transactions genuinely; 3) transactions with minimum score go to next round, others are rejected; 4) the final round requires a minimum of 80% agreement on the UNL of server *s*. Transactions that satisfy the requirement are added to the registry, which at the end is closed to compose a block of the chain. The main condition for which consensus is reached in all UNL is that each UNL is at least 40% overlapped with the others; if not, each UNL can reach its own consensus independently and without agreement.

Stellar Consensus Protocol (SCP). It is a quorum based Byzantine agreement protocol with open membership [25]. SCP is based on the concept of quorum, a set of nodes sufficient to reach the agreement, and quorum slices, subsets that can convince one particular node about the agreement. A single node can appear on multiple quorum slices. Slices and quorums are based on real-life business relationships between various entities, thereby leveraging existing trust. SCP is divided into phases: initial voting, accepting the vote, ratifying and confirmation. It reaches the global consensus in the entire system if quorums intersect. If a vote remains blocked, it uses a ballot-based approach to let the algorithm proceed.

Other used algorithms are Proof of Activity, Proof of Authority, Proof of Luck, Proof of Burn, Proof of Capacity, Proof of Importance [18], Raft, Tangle, and Algorand [26].

2.2.3 *Security and Countermeasures*

Security is a crucial aspect we can use to characterize blockchains. A (non-comprehensive) list of the most popular attacks [27] is shown in Table 3. Targeted blockchains and some possible countermeasures are highlighted.

Table 3: Some attacks against Blockchain.

Attack	Description	Target Blockchain	Countermeasures	Notes
DDoS	Exhausting network resources by sending a large number of requests.	All	Proof-of-Activity (PoA) protocol. Improve IoT devices security.	-
Majority (51%)	Hold the majority of hashing power to manipulate consensus.	PoW-based	Presence of observers nodes in the network	Also in PoS-based may occur if a node owns more than 50% of the total coins.
Double spending	Use the same cryptocurrency multiple times (e.g. by sending conflicting transactions in rapid succession).	PoW-based		-
Private key (Wallet) theft	Steal private key to steal cryptocurrency and identity.	Public	Two-factor security, Password-Protected Secret Sharing (PPSS)	In a private blockchain may be possible to track attackers behaviour.
Sybil	Create multiple nodes controlled by same entity.	Public	Limit number of output connections	In private blockchains, it can be avoided using heuristic rules.
Vulnerabilities in smart contracts	Various. I.e., send malicious transactions to steal currency.	Blockchain 2.0, (especially Turing-complete languages)	Not a clear answer	-
Attacks on cryptography	Breaking RSA encryption.	All	Switch to Quantum Resistant Ledgers	Currently not a problem, but can be in the future (quantum computing).

Some observations follow:

- The use of different types of consensus algorithms affects blockchain security (e.g., the 51% and double-spending attacks target PoW-based blockchains);
- Some attacks target specifically public blockchains. In some cases, private blockchains permit avoiding or mitigating the risk (e.g. for Sybil attack);
- The extension of capabilities in Blockchain 2.0 (e.g., the usage of smart contracts) leads to the introduction of software vulnerabilities.

As shown in the table, smart contract security (in particular for Turing-complete language) is a critical problem that does not have clear countermeasures. This Thesis focuses on this problem. Thus, the next section introduces basic concepts of smart contracts, focusing on Ethereum and its primary (and Turing-complete) language Solidity.

2.3 Smart Contracts

Smart contracts are one of the most important innovations of the second generation of the Blockchain. The basic idea is to execute computerized transactions automatically, depending on both external and internal conditions. Their diffusion has allowed the development of applications in different areas (e.g., financial, medical, insurance, gaming, betting). This section first presents some basic concepts dealing with smart contracts. Then, after reviewing the main platforms and use cases, it focuses on Ethereum and its primary language, Solidity. Finally, it deals with quality assurance.

2.3.1 *Definitions, Benefits and Limitations*

The first definition of smart contracts in 1997 [28] stresses the concept of contract term execution.

A smart contract is a computerized transaction protocol that executes the terms of a contract.

The concept has not received particular attention until the emergence of Blockchain and its support of high-level languages. In the original meaning, smart contracts met three main properties [12]:

- *Autonomy*: once a smart contract is executed, the parties involved are not required to remain in contact.
- *Self-sufficiency*: smart contracts are self-sufficient in managing resources (e.g., fundraising, providing services, purchasing resources for data storage or processing).
- *Decentralization*: smart contracts are stored into the Blockchain and executed synchronously by each node of the network (without the need for a central server).

More recently smart contract definition has been extended to *a persistent script stored into the Blockchain* [29]. This extension focuses on the *automated execution* on a Blockchain when some conditions are met [30]; moreover, it extends the concept of involved parties to involved entities (e.g., users, programs, systems).

There are many benefits of using smart contracts. Thanks to the *automated execution* and *self-sufficiency*, there is a substantial reduction of manual operations (and therefore the consequent risk of errors), elimination of duplicate work, time savings, speeding up the execution of operations, refund if conditions are (not) met. Thanks to *autonomy*, there is no need for intermediaries. In addition, *decentralization* allows for increased reliability.

2.3.1.1 Oracles

Specific conditions trigger the execution of smart contracts. If the condition is internal to the Blockchain, the smart contract can access it directly. If the condition is external, the smart contract must communicate with entities outside the Blockchain to continue execution. Since it cannot communicate directly, it relies on external services called *oracles*. There are several types of oracles [2], including:

- *Software Oracle*. It retrieves and extracts information online, providing it to the smart contract.
- *Hardware Oracle*. It allows to retrieve information from a CPS and make it available to the smart contract through sensors (e.g., RFID). Data are typically encrypted, and an anti-tamper mechanism is provided to increase security.
- *Inbound Oracle*. It acquires information from external sources. For example, a smart contract can automatically initiate an order

when the euro reaches a specific quote. The oracle provides the latter information.

- *Outbound Oracle*. It allows sending information from a smart contract to the outside world. An example is a smart contract that arranges for a garage door to open after the payment for parking.

- *Consensus-Based Oracle*. It consists of a combination of different oracles (even of different types). It is used to make decisions (typically based on data predictions).

2.3.1.2 *Issues*

There are some issues in using smart contracts:

- *Scalability*: the number of contracts and users cannot increase indefinitely. Increasing the number of contracts increases the number of transactions that each node executes (each node performs all transactions on the Blockchain to which it belongs). This requires an increase in hardware resources.

- *Legal*: smart contracts can define and perform legally binding contracts. Novel legal issues (e.g., contract formation, interpretation) can arise in this context. In addition, there is a lack of uniformity in international jurisdiction in resolving disputes that may arise from the performance of smart contracts [31].

- *Security*: smart contracts automated execution potentially endangers applications as they are immutable after storing them on a Blockchain. Design and coding faults and weaknesses in the smart contracts implementing the particular application can still result in exploitable vulnerabilities to malicious attacks despite the well-designed run-time environment.

This Thesis focuses on smart contract security.

2.3.2 *Platforms and Use Cases*

Ethereum, Hyperledger Fabric, Corda, Stellar, Nem, Neo, Eos are some of the most representative platforms that enable smart contracts. Table 4 compares them (extending [32]) in terms of language and its Turing completeness, consensus algorithm (Section 2.2.3), permissions (Section 2.2.1.2), execution environment and kind of application.

Table 4: Blockchain platforms that enable smart contracts.

	Ethereum	**Hyperledger Fabric**	**Corda**	**Stellar**	**Nem**	**Neo**	**Eos**
Language	Solidity Vyper	Java Golan Node.js	Java Kotlin	Python Javascript Golan	Java	Java Javascript Golan Python	C++
Turing completeness	Yes	Yes	No	No	Yes	Yes	Yes
Consensus algorithm	PoW PoS	PBFT	Raft	Stellar	PoI	BFT	BFT-DPoS
Permission	Public	Private	Private	Consortium	Public Private	Private	Public
Execution environment	EVM	Docker	JVM	Docker	Docker	NeoVM	WebAssembly
Application	General	General	General	Finance	General	Smart Economy	General

There are languages developed for specific platforms (e.g., Solidity, Vyper) or general-purpose languages (such as Java, Javascript, Python, C++). Turing completeness allows the development of smart contracts with greater expressiveness at the expense of greater susceptibility to attacks. PoW is computationally intensive when BFT-like consensus algorithms are network intensive. Other algorithms have intermediate characteristics, as highlighted in Section 2.2.2. Smart contracts are used in both types of public and private blockchain. The execution environment covers virtual machines and docker (reducing the overhead at the expense of application isolation). The types of applications that can be developed are general or specific, related to financial or smart economy sectors.

Smart contracts have a broad spectrum of applications that grows over time. A non-exhaustive list includes [31], [32], [33]:

- *Finance and banking*: capital markets and investment banking, commercial and retail banking, securities, insurance, trade finance, prediction markets.

- *Management*: digital properties and rights management, organizational management.
- *Public sector*: E-voting, personal reputation systems, smart property exchange.
- *Internet of Things and CPS*: energy, healthcare, supply chain, intelligent transportation systems.

2.3.3 *Ethereum and Principles of Solidity*

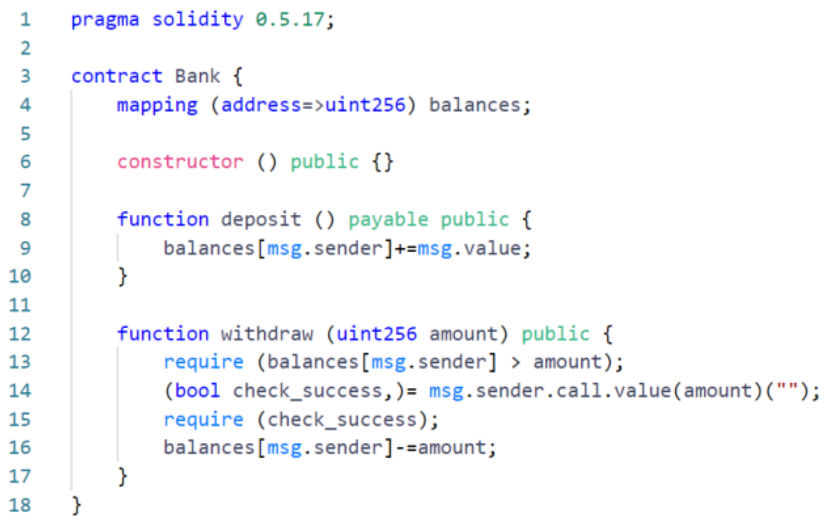
Ethereum is the second platform for market cap [34] after Bitcoin[2]. This section introduces some concepts related to Ethereum and its primary language, Solidity.

The *Ethereum Virtual Machine* (EVM) is the execution environment of Ethereum: each execution changes the EVM state [35]. Ethereum has two kinds of accounts: *externally owned accounts* and *contract accounts*; both have a *balance* field in *Ether* (the native currency of Ethereum). The first kind of account represents a user account, controlled by its private key; the second is a smart contract account, and its code controls it. A user can send a *transaction* (signed data package) to other accounts: if the receiver is a contract, it activates its code *executing it* into the EVM. Other contracts can trigger the code execution of a contract by *messages* (function calls). Blocks of the Blockchain contain all transactions and the related EVM state.

The *gas* represents fees to be paid for computations in Ethereum. Executing a transaction requires *computational steps* and then fees. Every transaction contains a *recipient*, a *sender* (identified by a signature), a *startgas* (maximum number of computational steps), and a *gasprice* (the fee the sender pays for each unit of gas). The product of startgas by gasprice is the *maximum fee* (in the units of Ether) paid to the *miner* processing the transaction. If the transaction terminates successfully, the miner returns unused gas to the sender.

The primary high-level language used to develop Ethereum smart contracts is Solidity [36]. C++, Python, and Javascript influenced Solidity. It has a similar programming structure to traditional languages, such as several types of variables, branching instructions, and assertions.

[2] Data refer to December 2, 2021.



```
1   pragma solidity 0.5.17;
2
3   contract Bank {
4       mapping (address=>uint256) balances;
5
6       constructor () public {}
7
8       function deposit () payable public {
9           balances[msg.sender]+=msg.value;
10      }
11
12      function withdraw (uint256 amount) public {
13          require (balances[msg.sender] > amount);
14          (bool check_success,)= msg.sender.call.value(amount)("");
15          require (check_success);
16          balances[msg.sender]-=amount;
17      }
18  }
```

Figure 4: Sample of Solidity smart contract.

The core of a smart contract consists of one or more *logic contracts* and optionally *libraries* and *interfaces* containing *state variables* and *functions*. Functions can execute instructions, interact with other contracts and modify state variables. Variables can have different main types (e.g., *boolean*, (*unsigned*) *integer*, *address*) and some derived ones (e.g., *structures*, *enum*). Each change of a state variable is saved permanently into the Blockchain. Moreover, a *modifier* changes the visibility of a function and its capability to receive Ether. Besides, functions receive the detail of the transaction. Once deployed into the Blockchain, the contract gets its address, a *constructor* (a special function) initializes its variables, and its code becomes immutable.

For a better understanding, Figure 4 presents a simple smart contract. The smart contract starts with a Solidity directive (at line 1) that identifies the release of the compiler to be used. A logic contract defines the program's core (at line 3). The Bank contract's state consists of the variable balances, that maps a variable of type address to an uint256 (unsigned integer with 256 bits). The contract has two functions: deposit (at line 8) and withdraw (at line 12).

The function deposit is payable: it can receive Ether. In particular, it receives an amount from the caller through msg.value, added to the balance of the caller at line 9 (available in msg.sender address).

The function withdraw permits to withdraw part of the address balance. At first, the requirement statement (at line 13) checks that the sender has enough funds in the Bank. In case of a positive check, withdraw sends Ether through the primitive call using the caller address. Called address can represent a contract: it can perform arbitrary actions (within the gas limits). The result of the call is stored on check_success. If check_success has true value, it passes the requirement at line 15, and the caller's balance is subtracted. This example contains a reentrancy vulnerability [3]. At line 14, the function withdraw transfers the control to the caller before subtracting the amount. This action permits the caller a continuous call of the withdraw function till the exhaustion of funds of contracts.

2.3.4 *Quality Assurance*

Quality assurance (QA) is one of the most tradition-richest fields in software engineering. The ISO/IEC 25000 family of standards defines system and software quality requirements and evaluation at the behavioural level, including extra-functional properties. The new ISO/IEC 5055:2021 standard [6] measures source code quality in an automated way based on an estimation of the complexity of the code under test and an empirical set of antipatterns, i.e., typical weaknesses causing failures.

This way, our basic assumption is that most weaknesses and resulting vulnerabilities in Solidity are similar to those in conventional programming languages, potentially appearing in a specific form.

Software security engineering has solid empirical foundations from a well-organized and maintained process of vulnerability data acquisition, abstraction, and generalization. The *Common Vulnerabilities and Exposures* (CVE) database [37] collects, defines, and catalogues publicly disclosed cybersecurity vulnerabilities, i.e., weaknesses in software (and hardware) components that, when exploited, spoil the security of the system. The *Common Weakness Enumeration* (CWE) [38] classifies weaknesses as root causes of vulnerabilities into a hierarchical taxonomy; furthermore, each CWE list item highlights the mode of introduction, expected consequences, and potential mitigations.

The hierarchy starting at the *Variant* level and continuing with the *Base* (typically related to a particular product, language, or technology) gradually eliminates the implementation details. The highly abstract top two categories of weaknesses in the hierarchy (*Class* and *Pillar*) are already independent of any specific language or technology. At the topmost level of abstraction, *Pillar* is a concept used to group weaknesses that share common characteristics. The notion of *Class* is abstract enough to be implementation independent but close enough to define functional and/or structural antipatterns leading to weaknesses.

CVE has more than 160000 entries currently, but only a few hundred are related to Blockchain technologies. However, the close relation of Solidity to traditional programming languages justifies the reuse of CWE classes for faithful fault modelling.
The CWE guides classification of vulnerabilities into the US government repository entitled *National Vulnerability Database* (NVD) [39] used for vulnerability management, security measurement, and compliance in a general technology context.

While CWE has a primary security focus, it is appropriate to describe weaknesses considering other extra-functional properties. For instance, as mentioned before, ISO/IEC 5055:2021 measures the reliability, performance, and maintenance of a code using CWE patterns.

This Thesis focuses on the aspect of security; accordingly, with definitions of Section 2.1, we refer to vulnerabilities as *exploitable weaknesses*.

2.4 Static Analysis

The main types of code analysis are *static* analysis, *dynamic* analysis, and *formal* verification. Static analysis inspects the code without executing it, searching for vulnerable patterns in the code structure. The dynamic analysis examines programs as they run in a run-time environment, acting like an attacker looking for vulnerabilities by providing malicious code or input to functions. Finally, the formal analysis uses theorem provers or formal methods to verify a program's specific properties, such as functional correctness.

This section focuses on static analysis, providing some introductory concepts and some characteristics of static analysis tools.

2.4.1 *Introduction*

Static analysis (SA) is one of the most significant and widely used types of code analysis. It inspects the code without executing it.

At first, it extracts an abstract model of the code under evaluation (typically an AST = abstract syntax tree or CFG = control flow graph). Then it can retrieve several types of information, including:

- potential vulnerabilities in the code over the model by pattern matching for weaknesses (antipatterns);
- dead code: code paths that cannot be reached;
- code anomalies;
- information for code optimization.

Software developers who use static analysis tools (referred simply as static analysis) can benefit from the facts produced by the analysis to further understand, evaluate, and modify the associated code.

SA has some advantages on dynamic analysis [40]. An advantage of static analysis is to operate on all possible execution branches in a program. On the contrary, the dynamic analysis only accesses the path of the code currently running. However, dynamic analysis manages to get information such as the location of the data in the memory of the program that is executed, while static analysis could only guess it.

SA is ubiquitous in modern software engineering. Popular examples include security flaws and defects scanners such as Klockwork and Coverity; programmer error detection tools such as scan-build, an analysis tool aimed at C, Objective-C, C++, and Swift; code formatters such as Python's black; user-oriented editor tools such as Rust's rust-analyzer.

There are also some limits to static analysis. No algorithm can determine whether the program terminates or loops indefinitely given a source code and all its possible inputs (the halting problem [41]). Most properties checked by static analysis are equivalent to the halting problem [40], [42]; thus, static analysis problems are *undecidable* in the worst case [43]. Static analysis approximates the program behaviour; however, this approximation is useful in practice.

2.4.2 *Characteristics of Static Analysis Tools*

This section describes the characteristics of SA tools used throughout the Thesis. The main characteristics are their input format, the internal representation extracted from the code, and the analysis methodologies applied. Some concepts will be further addressed in specific chapters.

2.4.2.1 The input

SA has the code of the smart contract to be checked. We distinguish here two possibilities:

- *Bytecode* is a list of compiled instructions executed in an Ethereum Virtual Machine (EVM);
- *Source code* refers to the smart contracts high-level programming language (e.g., Solidity).

2.4.2.2 The internal representation

The internal representation is about the abstract model extracted from the code for the analysis. Alternatives here are the *abstract syntax tree* (AST) or the *control flow graph* (CFG):

- AST extracts an abstract representation of the source code by lexical and syntax analysis.
- CFG is a directed graph representing the program flow derived from the AST or the bytecode. The basic blocks of a program serve as nodes. An arc connects node A to node B if block B may get executed immediately after block A. The arc labels represent the condition of the path execution.

2.4.2.3 Methodologies

Methodologies represent the algorithmic approach that tools use to analyze smart contracts for identifying vulnerabilities:

- *Decompilation* (DEC) transforms the bytecode into a language at a higher abstraction level (like an intermediate or Solidity-like language) to enhance the code's readability.
- *Disassembly* (DIS) transforms the EVM bytecode into an assembler language divided into blocks and assigns labels (e.g., to jump destinations and addresses).
- *Symbolic execution* (SE) uses symbols instead of real values of variables, based on determining the path code's reachability through constraints controlled by *Satisfiability Modulo Theories* (SMT) solvers.

- *Taint analysis* (TA) follows information flows generated from an information source. Initially, only deriving data from the source are considered contaminated. The method keeps track of how this taint propagates (it can happen, e.g., through assignment operation).

2.4.3 *Static Analysis for Vulnerability*

Static analysis is widely used to discover vulnerabilities in the early stages of the software life cycle. Despite its incomplete fault coverage, it can cover 100% of the code at a low cost [44].

Checking a program under test may result in a successful test run or a processing failure - an improper or incomplete test run with partial (or no) diagnostic outcome -. A successful test run may result in two different outcomes depending on whether the tool identified a vulnerability or not. A positive result (P - *positive* – the tool identified a vulnerability) can be a *false positive* (FP – wrong detection of a non-existing vulnerability) or a *true positive* (TP – correct detection of an existing vulnerability). Clearly, this is only a partial view as a negative result (N – *negative* - the tool did not identify a vulnerability) can be a *true negative* (TN – correct assessment of no vulnerability) or a *false negative* (FN – missed detection of an existing vulnerability).

We explicitly consider false positives and false negatives. In the previous section, we treated the issue of undecidability. Because of this problem, static analysis cannot be free of false positives. Moreover, false positives cause a big problem for all code analysts: a high rate of false positives forces a very high expenditure of time and resources for their detection (to distinguish them from true positives). Reducing the number of false positives takes resources and time [42]. The resulting tradeoff is very subtle: if the analysis is fast, it is likely to report many false positives. Conversely, an analysis is unlikely to finish in a reasonable time for large programs.

On the other hand, a high number of false negatives is dangerous because it leaves doors open for subsequent successful attacks. False negatives can occur for at least two reasons: the analysis is marred by unwarranted assumptions (e.g., not taking into account that malloc can return null), or the analysis does not consider all possible execution paths in the program and it is incomplete. The risk of having false

Table 5: Comparison between main vulnerability-related works.

Study	**Vulnerabilities**	**Grouped by**	**Propagations**
Chen et al.	26 (21)	Status	No
Atzei et al.	12 (11)	Solidity, EVM, Blockchain	No
Dika et al.	22 (15)	Solidity, EVM, Blockchain	No
Mense et al.	22 (16)	Solidity, EVM, Blockchain	No
Hasanova et al.	21 (18)	No group	No
Praitheesan et al.	15 (10)	Exploitability to attacks	No
Bartoletti et al.	Ponzi schemes	Type of Ponzi schemes	No
This Thesis	32	CWE classes (10)	Yes

negatives can be reduced by using multiple tools simultaneously (thus increasing the number of false positives).

2.5 The State of the Art

Several studies investigate smart contract vulnerabilities and their systematization and static analysis applied to smart contracts. The following state of the art review includes the most relevant papers as well as online resources available so far, mainly concentrated on Ethereum and its primary language, Solidity.

Table 5 highlights the relations between our systematization and some relevant previous studies by specifying the number of treated vulnerabilities, paired with the number of coinciding ones with our work, the nature of the classification and the analysis of the vulnerability propagations.

Atzei et al. [45] have, for the first time, deeply analyzed vulnerabilities, providing a taxonomy. They analyzed 12 kinds of vulnerabilities and linked 9 of them with different attacks. They identify two main reasons that make the smart contracts error-prone in Ethereum: the first reason is the Javascript-like nature of the Solidity language; the second one is the dissemination of the documentation in different sources. Their main focus was to organize a vulnerability taxonomy and check the effectiveness of attacks. Our work, differently, systematizes the vulnerabilities (using 11 of the 12 Atzei's vulnerabilities) in a non-language-related way and highlights propagation among them.

Dika et al. [46] proposed an up-to-date taxonomy of 21 vulnerabilities, grouping them based on their occurrence location: Solidity language, Ethereum Virtual Machine (EVM), or Blockchain level. This work also conducted experiments with some tools providing 21 vulnerable smart contracts and analyzing results.

The study of Mense et al. [47] used taxonomy to compare tools in discovering issues (based on their research paper). More than this, they analyzed an attack for providing a secure smart contract development.

Hasanova et al. [48] first investigated Blockchain vulnerabilities at the consensus level and types of potential attacks. Then it focused on 20 vulnerabilities smart contract related, highlighting adverse effects and providing possible countermeasures.

Praitheesan et al. [49] first analyzed vulnerabilities and their detection methods, then focused on the attacks that caused severe losses; finally, they classified the analysis methods into three categories (static, dynamic, and formal analysis).

Bartoletti et al. [50] presented a comprehensive survey of Ponzi schemes on Ethereum, analysing their behaviour and their impact from various viewpoints.

More recently, Chen et al. [51] presented a survey of 44 vulnerabilities, investigating the application layer, data layer, consensus layer, network layer, and Ethereum environments. Based on these aspects, causes, attacks, and exploit consequences are highlighted. Focusing on the application layer (which we are interested in), they propose a taxonomy of 26 main vulnerabilities (application related), grouping them by status (e.g., Smart contract programming, Solidity language and toolchain, Ethereum design and implementation) and highlighting causes, attacks, and consequences.

Emanuelsson and Nillson [42] performed one of the first comparisons among static analyzers to handle industrial applications.

McLean [52] focused on open source tools analyzing several programming languages (e.g., C, C++, Java). It performed an analysis of a pre-identified set of vulnerabilities, providing findings and uncovered vulnerabilities.

Table 6: Main static analyzers for Solidity.

Tool	**Input**	**Availability**
E-EVM	Bytecode	https://github.com/pisocrob/E-EVM
Erais	Bytecode	https://github.com/teamnsrg/erays
ETHBMC	Bytecode	https://github.com/RUB-SysSec/EthBMC
EtherTrust	Bytecode	https://www.netidee.at/ethertrust
EthIR	Bytecode	https://github.com/costa-group/EthIR
eThor	Bytecode	https://secpriv.wien/ethor/
GasChecker	Bytecode	No
Gasper	Bytecode	No
HoneyBadger	Bytecode	https://github.com/christoftorres/HoneyBadger
KEVM	Bytecode	https://github.com/kframework/evm-semantics
MadMax	Bytecode	https://github.com/nevillegrech/MadMax
Maian	Bytecode	https://github.com/ivicanikolicsg/MAIAN
Manticore	Bytecode	https://github.com/trailofbits/manticore
Mythril	Bytecode	https://github.com/ConsenSys/mythril
Octopus	Source code	https://github.com/pventuzelo/octopus
Osiris	Bytecode	https://github.com/christoftorres/Osiris
Oyente	Bytecode	https://github.com/enzymefinance/oyente
Porosity	Bytecode	https://github.com/comaeio/porosity
Rattle	Bytecode	https://github.com/crytic/rattle
Remix	Source code	https://github.com/ethereum/remix-project
SASC	Source code	No
Scompile	Source code	No
Securify	Bytecode	https://github.com/eth-sri/securify
Securify2	Bytecode	https://github.com/eth-sri/securify2
SIF	Source code	https://github.com/chao-peng/SIF
Slither	Source code	https://github.com/crytic/slither

Table 6 (continued): Main static analyzers for Solidity.

Tool	**Input**	**Public availability**
SmartCheck	Source code	https://github.com/smartdec/smartcheck
SmartEmbed	Source code	https://github.com/beyondacm/SmartEmbed
SmartInspect	Bytecode	No
SmartBug	Source code	https://smartbugs.github.io
SolAnalyzer	Source code	Yes
SolGraph	Source code	https://github.com/raineorshine/solgraph
SolHint	Source code	https://github.com/protofire/solhint
SolMet	Source code	https://github.com/chicxurug/SolMet-Solidity-parser
solc-verify	Bytecode	https://github.com/SRI-CSL/solidity
Vandal	Bytecode	https://github.com/usyd-blockchain/vandal
teEther	Bytecode	No
Verisol	Source code	https://github.com/Microsoft/verisol
Zeus	Source code	No

More recently, Nunes et al. [136] proposed a benchmark for web application static analyzers based on different criticalities levels, providing a general approach in benchmarking when different scenarios have to be analyzed.

Furthermore, the research into applying static analysis to detect Ethereum smart contracts' vulnerabilities increased significantly after the first infamous exploits in 2016 (e.g., DAO [3], starting from Oyente [54]. Main static analyzers developed explicitly for Solidity are summarized in Table 6.

Di Angelo et al. [55] independently analyzed 27 tools (static and dynamic) in terms of the method used, maturity, availability, and detection aspects. More recently, Vacca et al. [56] proposed a survey on Blockchain technologies in which they list several tools for analyzing smart contracts. Through the analysis of related papers, the techniques, datasets used and main results are highlighted.

Tikhomirov et al. [74] divided 20 kinds of smart contract bugs (we identify 15 vulnerabilities) into four groups (security, functional, operational, and developmental), analyzing several bugs. Then, they proposed a SA tool that uses an intermediate representation to detect them. Finally, a comparison with other tools is performed.

Durieux et al. [57] analyzed two datasets: the first one composed of more than 47.000 contracts, retrieve tools statistics, and the second one consisting of 69 contracts to deep analyze vulnerabilities. The set of vulnerabilities in their focus is related to the DASP repository. First, they evaluated the tool precision; in the large dataset, they identified the distribution of positives and analyzed the contract analysis time.

Parizi et al. [58] conducted an experimental assessment of static smart contracts security testing tools. They tested Mythril, Oyente, Securify, and Smartcheck on ten real-world smart contracts. Concerning the accuracy of the tools, Mythril was the most accurate.

Pinna et al. [59] performed a comprehensive empirical study of smart contracts deployed on the Ethereum blockchain to overview smart contract features, such as type of transactions, the development community's role, and the source code characteristics.

Ghaleb et al. [60] implemented SolidFI, a systematic method for automatically evaluating smart contract analysis tools using fault injection. They inserted a selected set of bugs in all valid identified locations. Akca et al. [104], by injecting a single bug into the contract code, compared the effectiveness of their static analyzer with some other tools.

Zhang et al. [61] focused on analyzing several bugs (among them, we can identify 20 vulnerabilities), grouped in 9 categories and based on the IEEE framework. It identified bugs that some tools do not cover. It provided a benchmark and based its analysis on precision and coverage.

More recently, Dias et al. [53] studied the effectiveness of the three SATs on a set of defects, classified according to the Orthogonal Defect Classification, as defined in [62].

Concerning software security, it is not always possible to analyze all vulnerabilities that have escaped detection; it is crucial to prioritize their analysis.

Eschelbeck [63] analyzed the problem of prioritizing software vulnerabilities within the vulnerability management process. The work provided general observations on the importance of patching the vulnerabilities with the highest severity. In a first step, Liu et al. [64] compared different prioritization methods using vulnerabilities extracted from the CVE database. In a second phase, vulnerabilities were grouped by type (CWE-based); the experiments improved scoring quality using vulnerability type.

Referring to previous works, some focuses on bugs (e.g., [60], [61], [53]); others compare static and dynamic analysis (e.g., [61]). In addition, some works analyze a small set of vulnerabilities (e.g., [57], [58]); others use a small set of tools (e.g., [57], [58], [53]), or provide no ground truth (e.g., [53], [57]).

Below are some of the main problems and limitations identified in state of the art, divided into three main key points:

a) Vulnerability systematization:

- Analyses are too tied to the language construct;
- Previous classifications either do not abstract from a specific Solidity release or do not capture the behaviour of vulnerabilities;

b) Assessment of the smart contract security:

- Previous analyses do not focus specifically on security;
- There is a lack in identifying how tools are built (how tools perform on the vulnerabilities they decided to target);
- Tool performances are investigated in a small set of vulnerabilities;
- There is a general lack of in-depth analysis of the tool detection capabilities in the different vulnerability classes;

c) Improvement of the smart contract security

- There is a lack of analyzing how to improve tool capabilities;
- State of the art does not analyze how tools can be combined to increase smart contract security;
- Previous works do not identify the most critical types of undetected vulnerabilities;

- The location (physical position) of vulnerabilities in Solidity smart contracts is not investigated.

Currently, there are no clear state-of-the-art answers on determining the assessment and improving smart contract security. Problems are mainly related to the rapid evolution of Solidity language, the consequent sparseness of vulnerability and the immaturity of checking tools.

2.6 APPROACH

Traditional languages (e.g., C, Java) have a long tradition of using SA. The broad spectrum of available tools raised the need to evaluate and compare them in a benchmark-styled way.

The main objective of end-users in using testing technologies is to ensure a good quality of the code and the productivity of the development process. At the same time, estimation of the insufficiencies is a primary input to the tool developers in their product development strategy.

Guaranteed quality necessitates a high probability of detecting faults, while productivity needs effective diagnosis and localisation to support debugging.

Measures of the detection effectiveness of an SA tool primarily address fault coverage, i.e., the ratio of fault kinds covered vs all faults anticipated. The minimisation of security risks additionally considers the frequency of occurrence and the severity of impacts of individual vulnerabilities in a more fine granular calculation of the fault coverage.

Creating the benchmark input set for software testing tools underlies several requirements. The programs consisting of the input benchmarking set have to represent the anticipated faults rooting in and covering observed vulnerabilities. This way, the set of tool testing input programs results from extensive vulnerabilities collection, root-cause analysis, categorisation, and potentially generalisation (e.g., mapping to CE) process. Evaluating the vulnerability and diagnosis capabilities of the tools requires the avoidance of benchmark test programs incorporating multiple vulnerabilities to avoid potential interferences between them.

Note that the statistically valid estimation of the frequency of occurrence needs a sufficiently large scale database. The assessment of the severity characteristics requires expert analysis of the potential impacts of a vulnerability.

One of the first examples to standardise the evaluation of SA tools was the NIST Software Assurance Reference Dataset Project covering, resulting in Juliet Test Suites for the most widely used programming languages, like Java, C/C++, and C# [138].

The main resulting vital points are:

- It is possible to identify the trigger of the corresponding vulnerabilities and weaknesses, which permits the study of the coverage.
- Each language-specific Juliet Test Suite can rely on an extensive, well-organised historical vulnerabilities database. Juliet anticipates a set of Base or beyond the Base level weaknesses from the CWE hierarchy as the source of vulnerabilities and covers them by test programs. Using such low-level weaknesses as a basis assures a close correlation between the tests and vulnerabilities. At the same time, the CWE-benchmark mapping delivers the ground truth by construction.
- The availability of a considerable number of (patterns) candidates permits a selective choice for inclusion into the benchmark to reduce false diagnoses due to multi-vulnerability interferences without compromising the fault coverage.

While the sound engineering principles of Juliet lead to a wide practical use for traditional programming languages, their adaptation to evolving languages faces severe difficulties due to the lack of sufficiently extensive experience.

Solidity, as already stated, is a new and evolving technology with changes in the language specification, as typical for technologies in their early lifecycle. According to the novelty of Solidity, only a limited set of well-documented and analyzed faulty patterns is available. Moreover, the relatively rapid evolution of the language definitions eliminates the most prevalent ones by improving the safety of the language constructs offered to the programmer. No similar deep analysis of the remaining patterns has happened yet, as for the repositories of vulnerable code fragments in traditional languages. Accordingly, there is no clear ground truth related to faults and selectivity.

In Solidity, it is possible having very similar patterns with different impacts. The lack of an extensive base confines the options of composing the benchmark set, especially to comply with the requirement of selectivity. Accordingly, we used the natural language description of vulnerabilities in the selection process.

At the same time, in the case of Solidity, our experience indicates that the current solutions are not extraordinarily sophisticated and well-elaborated. Thus, we use a top-down approach for the reasons listed above and contrary to Juliet. We generalise typical patterns into high-level categories by referring to Classes or Pillars instead of going to the lowest level of the CWE hierarchy (using Bases or Variants). Using large aggregate types of weaknesses avoids statistically unjustified partitioning of a relatively sparse dataset. In addition, we also have to create the ground truth due to the inadequate evaluation and documentation of vulnerability patterns.

Our patterns for evaluation can not cover all the potential patterns; however, we provide a representative model of Solidity in terms of observed vulnerabilities and weaknesses extracted from them. That means we can not guarantee that each pattern base is covered, but we offer a rough granular characterisation of the individual SA tools.

3

VULNERABILITY TAXONOMY AND PROPAGATIONS

This chapter investigates the first research question we address: *How to overcome the language evolution?* (**RQ1**).

To do so, Section 3.1 provides an overview of the motivations, Section 3.2 determines the set of vulnerabilities to focus on, and Section 3.3 systematizes vulnerabilities and provides a Solidity fault model based on CWE. Finally, Section 3.4 highlights some vulnerability propagations.

3.1 Motivation and Open Challenges

To have a global view on the topic, we have examined several papers (e.g., [45], [46], [47], [51], [65]) that analyze and classify platform-related vulnerabilities, and well-known exploits (e.g., [3], [66], [67]). Solidity is a new language, and it evolves rapidly. Considering previous studies, we noticed a difference in the number and categorization of such vulnerabilities, depending on the language or the specific platform. The missing agreement led to user confusion and vulnerabilities proliferation, as well as a difficulty for researchers to compare them. Moreover, previous classifications either do not abstract from a specific Solidity release or do not capture the behaviour of vulnerabilities.

The main motivation of this chapter is to overcome the strict dependence of the previous categorizations on the language release. Moving beyond language evolution is the foundation for the next steps in our research. To do so, we first provide Solidity-specific vulnerabilities analysis, then we identify CWE as a basis to classify vulnerability in a language-independent way. Our opinion is that this classification may help software developers limit weaknesses explosion and their effects, and researchers compare vulnerabilities of other platforms. In addition, this systematization allows us to focus on the vulnerabilities propagations between different CWE-based categories.

The contribution of this chapter is twofold:

- We provide a Solidity fault model, systematising smart contract vulnerabilities in a language-independent way. This model is the basis for the following chapters.
- We emphasize some propagations among classes of vulnerabilities based on well-known exploited contracts.

3.2 Smart Contract Vulnerabilities

This section identifies the vulnerabilities addressed in this work. The focus is on vulnerabilities originating in the development process of smart contracts. Accordingly, the underlying run-time platform is considered only as a potential error propagation path. Existing surveys (e.g., [51]) and sites provide lists of vulnerabilities, which, unfortunately, are updated very frequently due to the relative novelty of the technology.

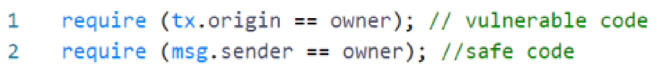
To provide a list of vulnerabilities, we ran a Google Scholar search using the keywords "Ethereum survey," "smart contracts analysis," and "smart contracts vulnerabilities." Additionally, we consulted the *Smart Contract Weakness Classification and Test Case* (SWC) registry [68], the Solidity documentation [36], and other referenced GitHub repositories to create a unified list of candidate vulnerabilities.

We then discarded from this list all the vulnerabilities already resolved at the language level from the Solidity release 0.5 onwards that we found on *Ethereum Improvement Proposals* (EIPs). EIPs are the standard Ethereum improvement; some EIPs mitigate or solve vulnerabilities (e.g., *Call-stack DepthValue*, fixed by EIP-150 [69] and *Under-priced Opcodes*, fixed by EIP-1884 [70]).

Finally, grouping vulnerabilities with similar or overlapped definitions resulted in a list composed of 32 items listed below in alphabetical order.

3.2.1 *Arbitrary Jump*

Solidity supports function types [71]; a variable of function type can be assigned with reference to a function with a matching signature. The



```
require (tx.origin == owner); // vulnerable code
require (msg.sender == owner); //safe code
```

Figure 5: Authorization through tx.origin: vulnerable and safe code snippets.

function saved to such a variable can be called just like a regular function [68].

Solidity does not support a generic pointer type; this does not (in general) allow changing this variable arbitrarily. However, if a smart contract uses specific assembly instructions (particularly mstore or assign operator), an attacker can execute random code instructions by changing the function type variable. This way, the attacker can alter, for example, state changes (directly impacting the data written in the Blockchain).

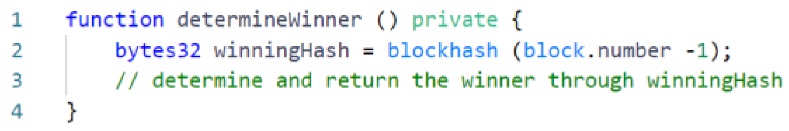
3.2.2 *Arithmetic Precision*

Solidity supports integers and (partially) fixed-point numbers. So, to represent floating points, integers must be used.

The 256-bit Ethereum virtual machine assembles types shorter than 32 bytes together in the 32-byte slot. This process affects the accuracy of any operation: for example, rounding is not correct if a division is performed before multiplication. If the result is used for critical operations, an attacker can force the use of data that alter the originally intended process.

3.2.3 *Authorization through tx.origin (Atx)*

Solidity's field tx.origin contains the address of the account that sent a transaction. Using it for authorization/authentication can make a contract vulnerable, potentially causing a fund loss. If an authorized account calls into a malicious contract, the latter can use its address to call the vulnerable contract and pass the authorization check (performed through tx.origin). This situation happens because tx.origin does not return the latest function call's sender, but the origin of the whole transaction, i.e., the authorized account [51]. The vulnerability can be prevented using msg.sender for authorization instead of tx.origin [47]. Two code snippets (vulnerable and safe) are shown in Figure 5.



```
function determineWinner () private {
    bytes32 winningHash = blockhash (block.number -1);
    // determine and return the winner through winningHash
}
```

Figure 6: Code snippet of blockhash usage.

3.2.4 *Blockhash Usage (BU) and Timestamp Dependency (TD)*

Blockhash usage [48] and Timestamp Dependency [45] vulnerabilities result from the usage, in critical operations, of blockhash and global timestamp variables, respectively. A malicious miner can manipulate such values. The code snippet in Figure 6 highlights that the use of blockhash value generates a BU vulnerability.

3.2.5 *Call to the Unknown (CU)*

Some Solidity's primitives for function invocation and Ether transfer (e.g., call, send) have the side effect of invoking the fallback function of the callee/recipient. This may lead to unexpected behaviours since an external portion of code is executed [45]. Mitigation is to avoid external calls whenever possible [47].

3.2.6 *Delegated Call to Untrusted Callee (DUC)*

The vulnerability was first observed in the Parity wallet attack [72]. The primitive delegatecall generates a message call that executes the code at the target address in the context of the calling contracts [51]. This way, an attacker can modify the state of the caller contract using the callee.

3.2.7 *DoS by External Contracts (EC):*

This vulnerability results from the dependence of conditional statements on external calls since the condition to continue the execution may never be satisfied [48].

3.2.8 *DoS Costly Patterns and Loops (CPL)*

Useless code and loops related patterns [51] could lead to a DoS situation when the gas needed to complete an execution exceeds the block gas limit. This issue is mainly caused by the use of unbounded operations (e.g., loops that depend on the function parameters that an attacker can manipulate). A possible mitigation for this vulnerability is to reduce the use of loops with a high or unknown number of iterations.

3.2.9 *Ether Lost in Transfer (ELT)*

Specifying a 160-bit address is required to transfer Ether. If the address is orphan (i.e., it is not associated with any accounts), the amount of money to be transferred will be lost, and it cannot be recovered. Developers that would avoid this problem have to ensure the correctness of the recipient's address with a check [45].

3.2.10 *Exceptions Disorder (ED)*

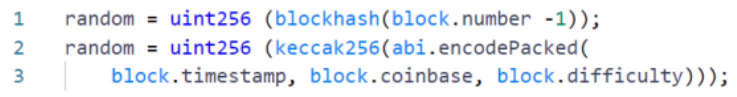
Exceptions can be raised in different situations, such as an out-of-gas exception, an error string exception, or a panic exception [45]. The inconsistency in the Solidity exception propagation policy can lead to implementation confusion, making contracts vulnerable. This issue can cause different effects and even allow attacks. Therefore, mitigation consists of checking any possible error condition.

3.2.11 *Freezing Ether (FE)*

This vulnerability results from the impossibility of a contract to send Ether while it can still receive it. It can result when a contract relies on no more available external code (for instance, after a `selfdestruct` operation) to perform the money transfer [51].

3.2.12 *Gasless send (Gs)*

The vulnerability results when the gas consumption for executing an operation exceeds the expected amount, raising an out-of-gas exception [45]. In particular, the function send does not specify the maximum amount of expendable gas for executing the recipient fallback function. Such an amount is fixed and likely to be reached when the fallback function contains many or expensive instructions. Considering that `send` does not propagate exceptions, the contract keeps the amount that should have been transferred.



```
random = uint256 (blockhash(block.number -1));
random = uint256 (keccak256(abi.encodePacked(
    block.timestamp, block.coinbase, block.difficulty)));
```

Figure 7: Generating randomness code snippet.

3.2.13 *Generating Randomness (GR)*

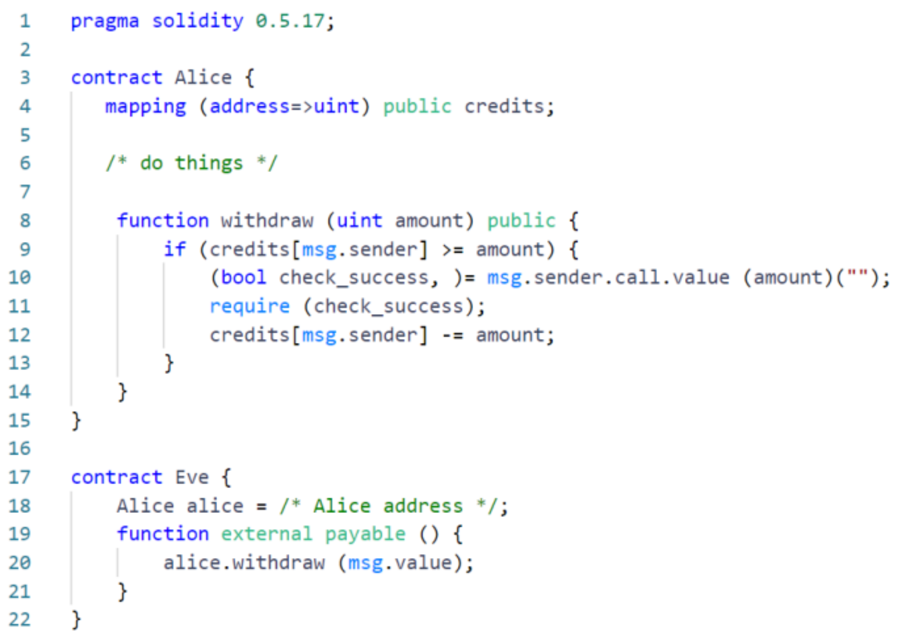
The execution of EVM bytecode is deterministic. To simulate non-deterministic choices, many contracts generate pseudo-random numbers, using the timestamp or other information (retrieved from global variables) about a block that will be added at a given time on the Blockchain. As an example, Figure 7 shows two different ways of generating randomness in such a way.
Since miners control the blocks of the Blockchain, a malicious one could craft a block to bias specific values [45], thus discovering the randomness and managing to manipulate events. A possible countermeasure consists of using an external source via oracles.

3.2.14 *Insufficient Gas Griefing*

A contract can accept data and use them for a sub-call to another contract. An attacker can provide enough gas for executing a transaction but not enough for the sub-call to succeed [68]. This way, the attacker can censor all transactions; the attacker has no direct benefit but causes damage to the victim of the attack [73].

3.2.15 *Integer Overflow/Underflow (IOU)*

A smart contract overflow occurs when a variable of type integer exceeds the maximum value that the type supports. Conversely, when the type is unsigned, decrementing a variable below the value zero will cause an underflow. Both an unmanaged arithmetic overflow and underflow can cause a software vulnerability [48], [51] (e.g., when the result value is used to manage resources or control the execution flow). However, the vulnerability can be mitigated by checking the operation's result.



```
pragma solidity 0.5.17;

contract Alice {
   mapping (address=>uint) public credits;

   /* do things */

    function withdraw (uint amount) public {
        if (credits[msg.sender] >= amount) {
            (bool check_success, )= msg.sender.call.value (amount)("");
            require (check_success);
            credits[msg.sender] -= amount;
        }
    }
}

contract Eve {
    Alice alice = /* Alice address */;
    function external payable () {
        alice.withdraw (msg.value);
    }
}
```

Figure 8: Reentrancy code snippet.

3.2.16 *Malicious Libraries (ML)*

Solidity permits building libraries that contracts can invoke. However, using libraries from untrusted sources may result in vulnerability by allowing the potential execution of malicious code [74].

3.2.17 *Missing Protection against Signature Replay Attack (MPRA)*

A smart contract can perform operations that need signature verification. In this case, protection against the Signature Replay Attack is needed (e.g., keeping track of all message hashes and only allowing new message hashes to be processed) [68]. A lack of this protection makes a contract vulnerable.

3.2.18 *Reentrancy (Re)*

This vulnerability occurs when a callee calls the calling function back before its completion. This call can lead to the repeated execution of functions designed to be executed only once [45]. The situation can easily occur when a contract sends Ether to another. In fact, in Solidity, a *fallback* function is called whenever a contract receives Ether or when a given not existing function is called. If the fallback function performs a callback, it will lead to a loop of calls that may drain the funds of the paying contract (as in the DAO attack [3]).

An example of a simplified version of the DAO attack is shown in Figure 8. If the malicious contract `Eve` calls the function `withdraw` of `Alice` (row 20) to get its deposited credit, its execution (row 10) invokes the `Eve` fallback function (row 19), generating a loop that will drain Alice's balance. This loop happens because the `Eve` credit update is performed after the payment (row 12) instead of before. A possible countermeasure to this vulnerability is to ensure that any external contract is called only once, updating appropriate variables before performing the call.

3.2.19 *Right to Left Override (RLO)*

Right-To-Left-Override is a special Unicode character (U+202E) that allows the use of right-to-left (RTL) characters within the text normally rendered left-to-right (LTR). For example, consider a smart contract: it is possible to insert an RTL character at appropriate points in the code to modify the behaviour in a hidden way. This way, the logic of the contract can be completely altered without a user being aware of it [68].

3.2.20 *Requirement Violation (RV)*

This vulnerability results from a violation of the requirements specified in a function for external input validation. Requirements violation indicates the presence of a vulnerability in the caller contract or an erroneous validation condition [68].

3.2.21 *Secrecy Failure (SF)*

Declaring a variable as private does not guarantee its secrecy because of the public nature of the Blockchain. Other contracts can not access the variable, but anyone can inspect published values and infer possible subsequent ones [51]. Using cryptographic techniques could mitigate the problem.

3.2.22 *Short Addresses (SA)*

When a contract’s invocation is performed, the corresponding transaction’s input field contains the callee function and all the call’s arguments. Each argument is encoded in 32 bytes automatically filled by the EVM with extra leading zeros if the length is shorter than allowed. The missing check of the validity of addresses by the EVM causes the vulnerability (EVM assumes that users always input 20-byte long addresses). For instance, `transfer(address to, uint fund)` allows the attacker to increase the amount of funds to gain [51] whether `to` is shorter than 32 bytes. The countermeasure to this vulnerability is checking the length of the transaction's input.

3.2.23 *Signature Malleability (SM)*

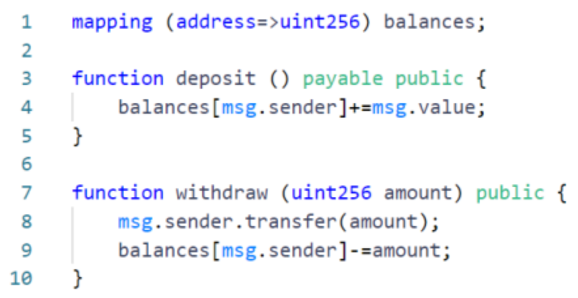
The Blockchain uses the public-private key mechanism (Chapter 2). Thus, users assume the uniqueness of the implementation of the signing mechanism in Ethereum smart contracts. However, the EVM specification allows the use of precompiled contracts (e.g., `ecrecover`) to retrieve the public key through a triple (`v, r, s`). According to [68], if the signature is part of the hash of a previously signed message, an attacker can modify the three parameters to create still-valid signatures (without possessing the private key).

3.2.24 *Transaction Ordering Dependence (TOD)*

This vulnerability arises when a contract relies on the order in which transactions are executed since miners decide this. There is no guarantee that the execution order matches the order in which transactions were requested, and this can affect the state of the dependent contract [51]. Using a pre-commit scheme [75] could mitigate the vulnerability.

3.2.25 *Typecasts (Ty)*

Receiving a contract as a function argument without checking its actual type represents a vulnerability. The Solidity type checker does not verify that the type is correct. Thus, calling a function of the received contract allows an attacker to execute arbitrary code. In fact, if a malicious contract contains a function having the same name as the invoked one, no error is raised, and the execution continues normally [45].



```
mapping (address=>uint256) balances;

function deposit () payable public {
    balances[msg.sender]+=msg.value;
}

function withdraw (uint256 amount) public {
    msg.sender.transfer(amount);
    balances[msg.sender]-=amount;
}
```

Figure 9: Unprotected Ether withdrawal code snippet.

3.2.26 *Unchecked Call Return Values (UV)*

Some Solidity functions for Ether transfer return the boolean value false in case of failure without raising an exception. Thus, a missing check of the return value results in a vulnerable condition [51], [68], leading to unexpected behaviours. The vulnerability can be mitigated by handling the return value.

3.2.27 *Unchecked send (Us)*

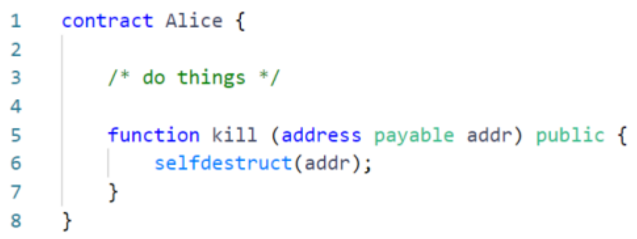
Unchecked send is a vulnerability produced by using the primitive function send. The primitive returns a boolean value; considerations are similar to Section 3.2.26. Therefore, the use of the send primitive is discouraged [36].

3.2.28 *Unprotected Ether Balance (UEB)*

Transactions that depend on the balance of a smart contract should be adequately protected. For example, suppose a contract performs a critical transaction when its balance reaches a certain amount (or it is greater than zero). Then, it is always possible to forcibly send Ether to a contract (e.g., using selfdestruct) by changing its balance even when there are mechanisms to disable receiving Ether. This way, a failure to protect access to the operation makes the contract vulnerable [68]; in the worst-case scenario, the vulnerability could lead to a DoS situation.

3.2.29 *Unprotected Ether Withdrawal (UEW)*

This vulnerability arises when malicious actors drain funds from a contract due to missing, insufficient, or wrong access controls [68]. The contract shown in Figure 9 allows other contracts to deposit and



```
contract Alice {

    /* do things */

    function kill (address payable addr) public {
        selfdestruct(addr);
    }
}
```

Figure 10: Unprotected selfdestruct code snippet.

withdraw Ether, keeping track of each contract's deposited balance. The code is vulnerable since the function withdraw allows a contract to get a specified amount of Ether even if it does not have enough in its balance. Because of the missing condition, a malicious contract could drain all the victim funds.

3.2.30 *Unprotected selfdestruct (Usd)*

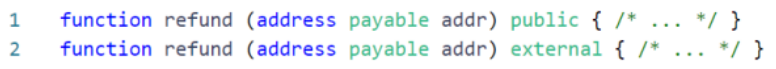
The use of the function selfdestruct without an appropriate check on the caller [51] may result in unintended destruction of the contract. For example, consider the code of Figure 10. If a non-authorized contract invokes the function kill passing its address as an argument, it will manage to destroy the contract Alice and gain Alice’s entire current balance.

3.2.31 *Visibility of Exposed Functions (VEF)*

An attacker could execute a function for arbitrary purposes when its visibility is wrongly defined. Consider the function definitions as shown in Figure 11. In both cases, the function visibility has been set incorrectly since external contracts can invoke the function that should only be called internally. The use of wrong function modifiers may allow unauthorized execution [51]. The effects can be various depending on the nature of the exploited function.

3.3 Vulnerability Systematization

This section deals with vulnerability systematization. It first reports on a more detailed description of CWE (introduced in Section 2.3.4) and



```
function refund (address payable addr) public { /* ... */ }
function refund (address payable addr) external { /* ... */ }
```

Figure 11: Vulnerability of exposed function code snippet.

then provides the Solidity fault model.

3.3.1 *CWE and Hierarchical Representation*

To better understand the systematization process (addressed in the next section), this section provides more details related to CWE.

The *Common Weakness Enumeration* (CWE) is a community-developed list (updated periodically) of common weakness types that involve security [38]. CWE community includes 58 members among some of the major technology companies (e.g. Apple, IBM, Microsoft), U.S. Government Institute/Agency (e.g. the National Security Agency, National Institute of Standard and Technology), security and cyber-security companies, and non-profit organizations.

According to [38], CWE proposes to:

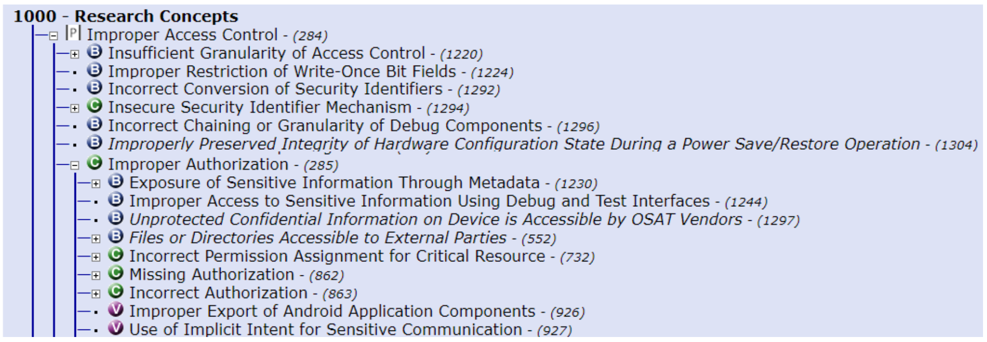
- have a common language for the weakness description, identification, mitigation;
- asses the coverage of tools (that target those weaknesses);
- prevent software vulnerabilities prior to deployment.

The *Common Weakness Enumeration* (CWE) classifies weaknesses as root causes of vulnerabilities in a hierarchical taxonomy. CWE provides three main hierarchical representations (*research concepts*, *hardware design*, *software development*) based on a specific point of view, others based on entries subset or specific domain or use case. For our purpose, we focus on the *research concepts* view that highlights:

- the abstraction of weakness behaviour, permitting to overcome the dependency on specific language and platforms;
- where and in which phase weakness appears in the code;

Each entry contains three elements that help in understanding how the vulnerability is relevant in a particular environment:

- The *introduction phase* identifies when the weakness is introduced in the software (e.g., architecture, design, implementation).
- *Common consequences* determine eight pre-defined technical impacts (*read-data, modify-data, DoS-unreliable execution, DoS-resource consumption, execution of unauthorized command, gain privilege/assume identity, bypass protection mechanism, hide activities*).



```
1000 - Research Concepts
—⊟ P Improper Access Control - (284)
   —⊞ B Insufficient Granularity of Access Control - (1220)
   —• B Improper Restriction of Write-Once Bit Fields - (1224)
   —• B Incorrect Conversion of Security Identifiers - (1292)
   —⊞ C Insecure Security Identifier Mechanism - (1294)
   —• B Incorrect Chaining or Granularity of Debug Components - (1296)
   —• B Improperly Preserved Integrity of Hardware Configuration State During a Power Save/Restore Operation - (1304)
   —⊟ C Improper Authorization - (285)
      —⊞ B Exposure of Sensitive Information Through Metadata - (1230)
      —• B Improper Access to Sensitive Information Using Debug and Test Interfaces - (1244)
      —• B Unprotected Confidential Information on Device is Accessible by OSAT Vendors - (1297)
      —⊞ B Files or Directories Accessible to External Parties - (552)
      —⊞ C Incorrect Permission Assignment for Critical Resource - (732)
      —⊞ C Missing Authorization - (862)
      —⊞ C Incorrect Authorization - (863)
      —• V Improper Export of Android Application Components - (926)
      —• V Use of Implicit Intent for Sensitive Communication - (927)
```

Figure 12: Hierarchy sample of CWE.

- *Potential mitigations* provide suggestions and recommendations on the countermeasures to be applied during the software development phases.

Figure 12 shows an example of hierarchy (extracted and adapted from [76]). The Pillar (*Improper Access Control*) has both Classes and Bases as sons (depending on the defined abstraction level); each Class could have a sub-hierarchy starting with other Classes, Bases or Variants. We highlight that each Pillar has its sub-hierarchy (independent from others) and that leaves are represented by Variants or Bases, depending on the represented level of detail.

3.3.2 *Solidity Fault Model*

This section systemizes the vulnerabilities addressed in this Thesis and identified in Section 3.2.

3.3.2.1 *Mapping vulnerabilities to CWE*

We have grouped the list of 32 vulnerabilities in a language-independent classification, in general abstract classes using a subset of the CWE-1000 Research Concepts [76] (based on abstractions of software behaviour).

As CWE is only a semi-formal taxonomy without formal semantics, mapping Solidity vulnerabilities into CWE is **manual**. Our methodology (introduced in Section 2.6) has its foundation in two main key points:

- identifying the CWE-ID level that best fits each vulnerability, placing it in a Pillar sub-hierarchy.

- finding *Classes* or *Pillars* (in the Pillar sub-hierarchy) that group multiple vulnerabilities of similar behavior.

Our first target is to identify the CWE-ID *Base* or *Class*-level weakness that is the best abstraction for each vulnerability. This process follows the *criteria for the best match* [137].

We consider *vuln* as a type of vulnerability that belongs to our taxonomy. To map the vulnerabilities onto CWE, we used the following method:

∀ *vuln*

1. We identified the main characteristics of *vuln* (e.g., the software's resources' exhaustion), extracting some keywords;

2. We identified the candidate shortlist CWE-IDs by performing an abstract *keyword* and *synonym* search in the CWE list.

3. An in-depth review of every element in this list led selecting the specific CWE-ID *Base* or *Class*-level (under the *Pillar* sub-hierarchy) to represent *vuln.*

Our final target is group vulnerabilities. Starting from the CWE-IDs identified in the previous step, we proceed bottom-up in the hierarchy until we find *Classes* or *Pillars* (in the *Pillar* sub-hierarchy) that group vulnerabilities with similar behaviour. Thus, we select them as our classification categories.

Consider two vulnerabilities as examples of the classification:

- *Gasless send* (Gs): it happens when a call invocation provides a limited quantity of gas to the callee, and the gas consumption for executing an operation exceeds the provided amount [45].

- *DoS costly Patterns and Loops* (CPL): a DoS situation can happen if the gas needed to complete an execution exceeds the gas block limit [51]. A potential cause is using an unbounded operation (e.g., loops that depend on a function parameter).

In the case of the example, the first step identifies CWE-400 (*Class*-level) to represent both vulnerabilities. By performing the second step, we have to select a class or a Pillar that groups the vulnerabilities. CWE-400 is still a Class and contains vulnerabilities with similar behaviour; thus, we select it as our classification category for Gs and CPL.

The classification of 32 vulnerabilities (Table 7) contains the acronym (Acr.), the full name, the reason for classification, the CWE category (specifying Class or Pillar), and finally, a short description.

3.3.2.2 Comparison between the fault model and the ISO/IEC 5055:2021 standard

As the next step, we check that the similarity between Solidity and conventional programming languages manifests in their respective fault model. We compare our proposed model for Solidity (10 CWE-IDs) and the ISO/IEC 5055:2021 standard (71 CWE-IDs) for conventional languages.

Note that the comparison of the fault models should cover both identities and similarities of the weaknesses in the two fault models. For instance, two weaknesses sharing a joint abstract ancestor in the CWE hierarchy indicate a common root cause manifested in different forms due to the peculiarities of the different programming languages.

12 CWE-IDs in the ISO list are irrelevant as Solidity has no similar language constructs. 7 ISO CWE-IDs related to software obsolescence are irrelevant for our purposes. 20 ISO CWE-IDs are descendants of our Solidity model. One CWE-ID is identical in the ISO and Solidity lists. Two Solidity and ISO CWE-IDs have a joint ancestor in the CWE hierarchy. 29 ISO CWE-IDs are derivatives of the family containing our Solidity CWE-IDs. A detailed comparison is available in [124]. The model for *Solidity* contains *3 Solidity-specific CWE-IDs that extend the list* of the ISO standard:

- *CWE-330 (Use of Insufficiently Random Values)* is related to generating random numbers whose seed comes from a value stored in the Blockchain.
- *CWE-345 (Insufficient Verification of Data Authenticity)*, concerning the use of signature and invalid data.
- *CWE-284 (Improper Access Control)*, related to improper authorizations or controls that permit unauthorized operations and Ether manipulations.

Table 7: Mapping between vulnerabilities and CWE based classification.

Vulnerabilities		**Classification**		
Acr.	***Name***	***Reason for classification***	***CWE-ID***	***CWE description***
ELT	Ether Lost in Transfer	Missing address validity checks.	CWE-20 (Class)	*Improper Input Validation*: the software does not validate or improperly validates input data.
RV	Requirement Violation	Improper input validation conditions.		
SA	Short Addresses	Improper validation of the address length.		
Atx	Authorization through tx. origin	Improper authorization restriction using tx origin.	CWE-284 (Pillar)	*Improper Access Control*: the software does not restrict or incorrectly restricts access to a resource. It involves authentication, authorization, accountability.
UEW	Unprotected Ether Withdrawal	Improper access control in Ether withdrawal.		
Usd	Unprotected selfdestruct	Self-destruction with improper authorization checks.		
VEF	Visibility of Exposed Functions	Improper access control or authorization allows improper function usage.		
GR	Generating Randomness	Use of predictable random numbers.	CWE-330 (Class)	*Use of Insufficiently Random Values*: the software generates predictable values in a context that requires unpredictability.
MPRA	Missing Protection against Signature Replay Attack	Missing check or protection in data authenticity.	CWE-345 (Class)	*Insufficient Verification of Data Authenticity*: the software accepts invalid data, improperly verifying their validity or authenticity.
SM	Signature Malleability	Improper verification of data signature.		
Ty	Type Casts	Improper verification of data validity.		
CPL	DoS costly Patterns and Loops	Improper management of resources (gas) in pattern and loop execution.	CWE-400 (Class)	*Uncontrolled Resource Consumption*: the software does not correctly control the allocation of limited resources, permitting an attacker to exhaust them.
Gs	Gasless send	Improper check in the usage of gas using send.		
BU	Blockhash Usage	Blockhash usage in critical operations exposes to manipulation from miners.	CWE-668 (Class)	*Exposure of Resource to Wrong Sphere*: the software provides unintended actors with inappropriate access to the resource.
ML	Malicious Libraries	Inappropriate access to resources.		
SF	Secrecy Failure	Anyone can accede to a private variable.		
TD	Timestamp Dependency	Timestamp usage in critical operations exposes to manipulation from miners.		

Table 7 (continued): Mapping between vulnerabilities and CWE based classification.

Vulnerabilities		**Classification**		
Acr.	***Name***	***Reason for classification***	***CWE-ID***	***CWE description***
CU	Call to the Unknown	Low-level function calls can be unintended controlled from the resources of another sphere.	CWE-669 (Class)	*Incorrect Resource Transfer Between Spheres*: the software provides an unintended control over the resource importing (transferring) it from (to) another sphere.
DUC	Delegatecall to the Untrusted Callee	Low-level function calls can provide unintended control to a resource of another sphere.		
EC	DoS by External Contracts	External contracts can cause unintended control from a resource of another sphere.		
AP	Arithmetic Precision Order	Divide before multiply can lead to incorrect results.	CWE-682 (Pillar)	*Incorrect Calculation*: software performs a calculation that leads to incorrect or unintended results.
IOU	Integer Overflow or Underflow	Overflow or underflow can lead to incorrect results.		
AJ	Arbitrary Jump	Execution of unexpected instructions.	CWE-691 (Pillar)	*Insufficient Control Flow Management*: the software does not properly manage the program control flow, permitting to modify it unexpectedly.
FE	Freezing Ether	Modification of the program flow makes it impossible to send Ether.		
IGG	Insufficient gas griefing	Prevention of sub-call execution alters the program flow.		
Re	Reentrancy	A callee calls the function back before its completion.		
RLO	Right Left Override	The standard flow of the program is modified.		
TOD	Transaction Ordering Dependence	An attacker can artificially favor the execution of one transaction over another.		
UEB	Unexpected Ether Balance	Strict Ether balance assumptions cause an unexpected program flow.		
ED	Exception Disorder	Incorrect handling of Solidity exception propagation.	CWE-703 (Pillar)	*Improper Check or Handling of Exceptional Conditions*: the software manages improperly exceptional conditions.
Us	Unchecked send	Improper checks of exceptional conditions using send.		
UV	Unchecked Call Return Values	Missing checks of return values.		

The fault models for the general-purpose elements in conventional languages (ISO standard) and Solidity are similar, while in addition, Solidity has some Ethereum-specific faults. If the Solidity-relevant fault classes are identical to or descendants of the ISO standards, then the algorithms elaborated for traditional languages are reusable. Otherwise, new or extended algorithms are needed.

3.4 Propagation and Relations among Vulnerabilities

This section aims to emphasize some relations and propagations among the above-described vulnerabilities. Main information is extracted from most famous attacks (e.g., Dao [3], King of the Ether Throne [66], the Parity Multisig [72], Government [67]), and exploited smart contracts monitored by NVD. Furthermore, we include some code snippets and references to real contracts to support the explanation. Let us only note that the shown code (compliant to Solidity 0.5.x) may differ from the original one (Solidity 0.4.x): refer to Solidity breaking changes [77] for details.

Relations among vulnerabilities are shown in Figure 13 through circle intersections, where circles stand for vulnerabilities, each identified by its acronym. In particular, when a circle is contained in another, its corresponding vulnerability is a subset of the external one. Moreover, an arrow from circle A to circle B means that A may imply the presence of B under specific conditions. Rectangles identify CWE-IDs. After this premise, we describe some interdependencies and propagations.

First, let us consider the upper-left part of Figure 13. *Timestamp Dependency* (TD) and *Blockhash Usage* (BU) intersect since both are related to the usage of blockchain global variables controlled by miners. For example, the code snippet in Figure 6, highlights the use of `blockhash` value that generates a BU vulnerability: in fact, each global variable could be manipulated by miners. In addition, using a blochask or timestamp-dependent variable can lead to a GR vulnerability (that can be exploited as in [78]).

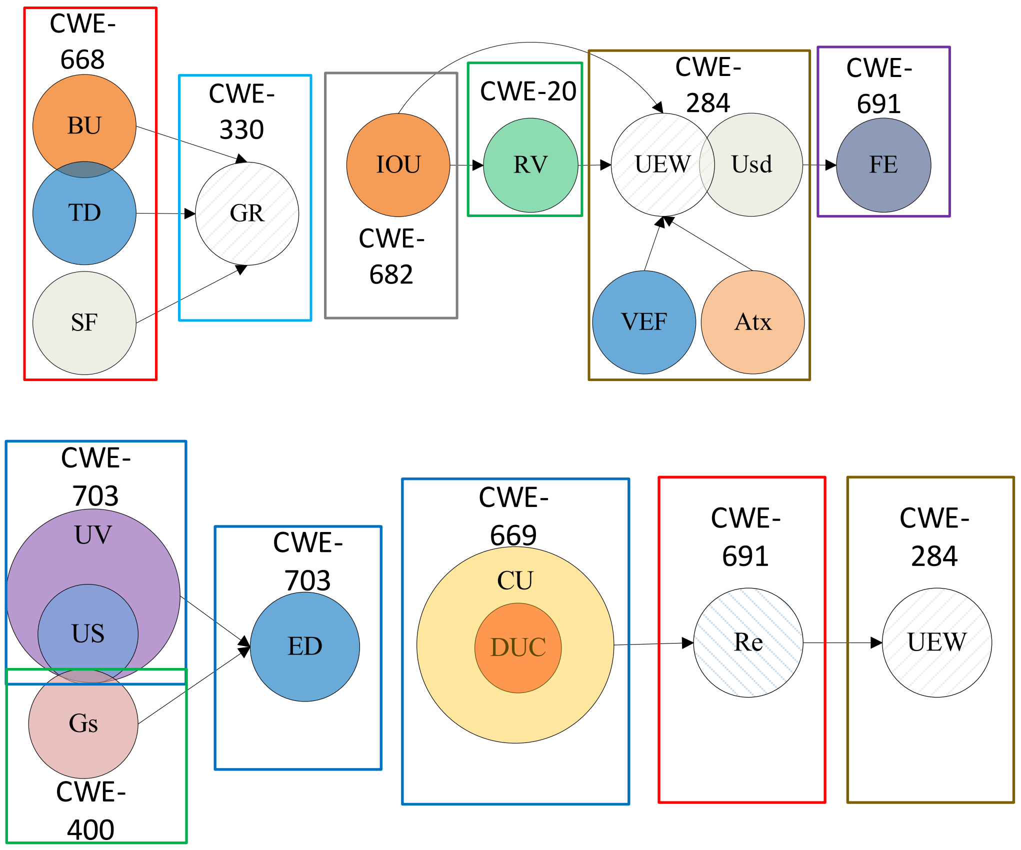


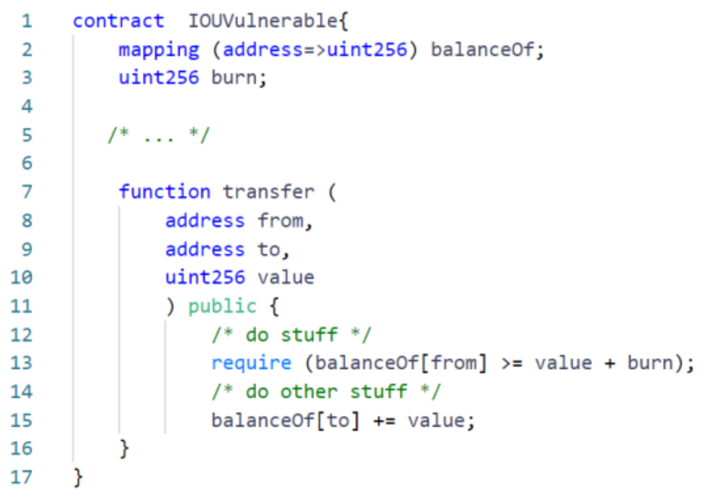
Figure 13: Vulnerability propagations.

Let us now consider the *secrecy failure* (SF). Making a variable private prevents other smart contracts from reading its value; however, the value of the variable is visible on the Blockchain. Whether a private variable is used as a seed of randomness, the secrecy failure could lead to GR (as in [79]).

As a global view, vulnerabilities that belong to CWE-668 (*Exposure of Resource to Wrong Sphere*) can propagate to CWE-330 (*Use of Insufficiently Random Values*). Exploiting the possibility to read secret data may allow an attacker to bypass protection mechanisms and guess the pseudo-random value.

The upper right part of Figure 13 shows the relationship between vulnerabilities of CWE-284 (*Improper Access Control*) and those of other classes.

Consider the chain *Integer Overflow or Underflow* (IOU) -> (that leads to) *Requirement Violation* (RV) -> (that leads to) *Unprotected Ether Withdrawal* (UEW). An example is shown in Figure 14. The RV is

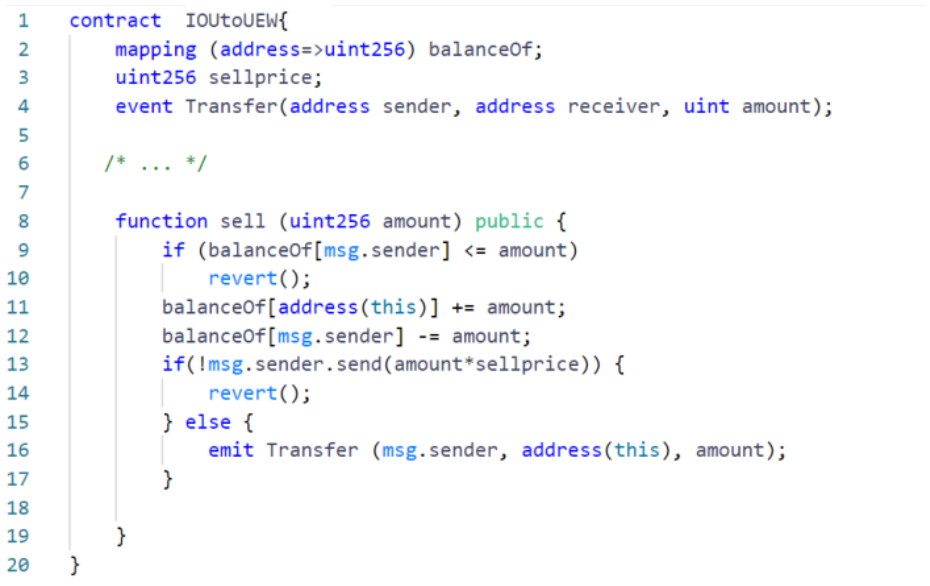


```
1   contract  IOUVulnerable{
2       mapping (address=>uint256) balanceOf;
3       uint256 burn;
4
5      /* ... */
6
7       function transfer (
8           address from,
9           address to,
10          uint256 value
11          ) public {
12              /* do stuff */
13              require (balanceOf[from] >= value + burn);
14              /* do other stuff */
15              balanceOf[to] += value;
16      }
17  }
```

Figure 14: Code snippet of IOU leading to RV.

caused (row 13) by the fact that an external contract may craft the function parameter value, leading to an integer overflow in the require clause (a real contract is shown in [80]). After the RV succeeds because of an overflow, the attacker may exploit the *UEW* vulnerability, gaining a potentially enormous amount of Ether (as shown in row 15 of Figure 14). On some occasions, as in Figure 15 (extracted from a [81]), an attacker (the contract's owner) could exploit an IOU to lead directly to a UEW.

A simple wrong function visibility setting that generates a *Visibility*



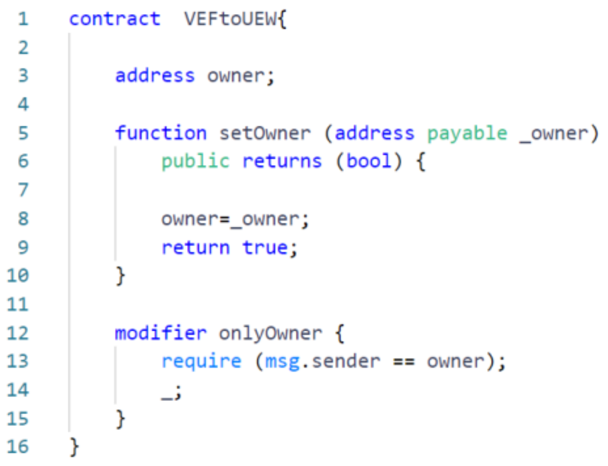
```
contract  IOUtoUEW{
    mapping (address=>uint256) balanceOf;
    uint256 sellprice;
    event Transfer(address sender, address receiver, uint amount);

   /* ... */

    function sell (uint256 amount) public {
        if (balanceOf[msg.sender] <= amount)
            revert();
        balanceOf[address(this)] += amount;
        balanceOf[msg.sender] -= amount;
        if(!msg.sender.send(amount*sellprice)) {
            revert();
        } else {
            emit Transfer (msg.sender, address(this), amount);
        }

    }
}
```

Figure 15: Code snippet of IOU leading to UEW.



```
contract  VEFtoUEW{

    address owner;

    function setOwner (address payable _owner)
        public returns (bool) {

        owner=_owner;
        return true;
    }

    modifier onlyOwner {
        require (msg.sender == owner);
        _;
    }
}
```

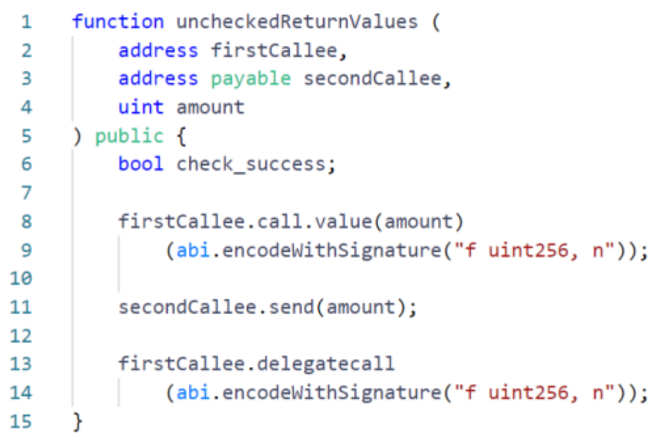
Figure 16: Code snippet of VEF leading to UEW.

of Exposed Function (VEF) is listed in Figure 16. If a user misuses the function, it may be possible for an attacker to withdraw Ether (UEW) illicitly. In the code snippet, the function `setOwner` has improperly `public` visibility allowing any user to modify the contract's owner, thus possibly causing the UEW (as in [82]). Also, the *Authorization from tx.origin* (Atx) vulnerability could propagate to a UEW with the same mechanism just explained.

The *Unprotected selfdestruct* (Usd) intersects with UEW because an improper access control causes both vulnerabilities; moreover, Usd could lead to a *Freezing Ether* (FE) if a contract is improperly destroyed (a famous related exploit is the Parity Multi-Sig [72]).

From a classification point of view, it appears that vulnerabilities that belong to CWE-682 (*Incorrect Calculation*) could lead to CWE-284 (*Improper Access Control*). In addition, improper controls could cause a loss of normal program flow.

Let us consider the bottom left part of Figure 13. The picture represents *Unchecked send* (Us) as a subset of *Unchecked Value* (UV) that includes primitive functions such as `call`, `delegatecall`, `send` (as shown in Figure 17). Moreover, the Us and *Gasless send* (Gs) originate from the `send` function (justifying the intersection in our picture). All these vulnerabilities (UV, Us, Gs) can lead to an *Exception Disorder* (ED): missing a check on a returned value of the involved functions implies the impossibility to manage an exceptional situation (an attacker may also provoke that): an example is the King of the Ether Throne [66].



```
function uncheckedReturnValues (
    address firstCallee,
    address payable secondCallee,
    uint amount
) public {
    bool check_success;

    firstCallee.call.value(amount)
        (abi.encodeWithSignature("f uint256, n"));

    secondCallee.send(amount);

    firstCallee.delegatecall
        (abi.encodeWithSignature("f uint256, n"));
}
```

Figure 17: Code snippet of Us, UV.

Let us then consider the bottom right part of Figure 13. The figure indicates *Delegatecall to Untrusted Callee* (DUC) as a subset of *Call to the Unknown* (CU). Using a function call to send Ether could cause the execution of the fallback function of a malicious attacker (*Reentrancy* – Section 3.2.18). Reentrancy, if exploited, could lead to the UEW (all contract funds are drained): an excellent example of this vulnerability propagation can be found in [72].

Considering this propagation from the classification point of view, we could infer that an *unintended control* over a resource could cause *an insufficient control* of the control flow, leading to an *access control* problem.

4 ASSESSMENT OF THE SMART CONTRACT SECURITY

This section deals with the assessment of smart contract security, addressing **RQ2:** *How can we evaluate the security of smart contracts by using static analysis to detect the most relevant vulnerability-related weaknesses?*

After selecting SA tools in Section 4.1, we provide the experimental settings and the methodology in Section 4.2. Section 4.3 investigates the testing performances of SA tools, and Section 4.4 deals with the validity and limitations.

4.1 Static Analyzers

This section describes the tools selection process, and a qualitative analysis identifying vulnerabilities not targeted for detection by the individual tools.

4.1.1 *Tools Selection*

At first, an extensive search of different sources such as research papers (e.g., [55], [56]) and other online resources (sites for developers of the Ethereum environment and GitHub's most referenced tools repositories) produced a shortlist of 38 candidate tools.

The final set of 9 tools (Table 8) selected for detailed analysis included only those fulfilling the following criteria:

- handling of contracts written in Solidity version 0.5 or higher;
- stand-alone tools targeting vulnerabilities detection;

Table 8: Selected tools.

Tools		**Analysis Methods**		
Name	***Release***	***Input***	***Internal representation***	***Methodology***
Securify2	Mai 2020	BC	CFG	DEC, DIS
Securify	Mai 2020	BC	CFG	DEC, DIS
Slither	0.7.1	SC	AST, CFG	TA
SmartCheck	2.1	SC	AST	DEC
Remix IDE	March 2021	SC	AST	Various
Mythril	0.22.17	BC	CFG	SE
Oyente	November 2020	BC	CFG	SE
Osiris	0.0.1*	BC	CFG	SE
HoneyBadger	0.0.1*	BC	CFG	SE

- analysis of smart contracts without user-defined properties or assertions;

- free public availability (for a white box analysis discussed later).

Input, internal representation and methodology refer to concepts described in Section 2.4.2. A * in the table indicates that specific options extend the support of the original Solidity tool to the 0.5 release (e.g., [83]).

For the sake of completeness, the list of tools excluded due to the violation of one or multiple selection criteria consists of E-EVM [84], Erays [85], ETHBMC [86], EtherTrust [87], EthIR [88], eThor [89], GasChecker [90], Gasper [91], KEVM [92], MadMax [93], MAIAN [94], Manticore [95], Octopus [96], Porosity [97], Rattle [98], SASC [99], sCompile [100], SIF [101], SmartEmbed [102], SmartInspect [103], SmartBug [57], SolAnalyzer [104], SolGraph [105], SolHint [106], SolMet [107], solc-verify [108], Vandal [109], Verisol [23], Zeus [110].

A short description of the selected tools follows:

- *Securify* (Sfy) uses antipatterns to decide if the software has unsafe behaviour, with the support of a domain-specific language [111].

Fallback functions, libraries, and abstract contracts are not supported. We used the main branch release [112].

- *Securify2* (Sfy2) represents a development of Securify taking a Solidity file as input and supporting only flat contracts. The tool decompiles the stack-oriented bytecode into an assignment-based form and transforms the code to DataLog. We used the release of the main branch [113].

- *Slither* (Sli) converts Solidity smart contracts into an intermediate representation called SlithIR [114]. We downloaded the 0.7.1 release from [115].

- *SmartCheck* (SmC)) identifies smart contracts vulnerabilities by searching specific source code antipatterns [74]. The tool converts the code into an XML syntax tree, and Xquery path expressions retrieve the vulnerable patterns. We used the master branch [116].

- *Remix-IDE* (Rmx) is continuously under development [117]. Based on different modules, it also performs a static analysis on that we focus. Transforming the code into an AST representation checks the software security by checking unsafe patterns. We installed the release of March 2021.

- *Mythril* (Myt) uses symbolic execution based on EVM bytecode for Ethereum and other EVM-compatible blockchains [118].

- *Oyente* (Oye) is a precursor in the field [54], and several other projects have used it as a starting and reference point. It uses symbolic execution. We used the master branch [120].

- *Osiris* (Osi) extends Oyente's fault model by integer overflows and underflows [119]. It combines symbolic execution and taints analysis. The version downloaded [121] extends Oyente 0.2.7.

- *HoneyBadger* (HoB) is another Oyente-based tool that employs symbolic execution and a set of heuristics to pinpoint specific vulnerabilities in smart contracts [122]; we used the release based on Oyente 0.2.7 [123].

4.1.2 *Preliminary Evaluation*

Which vulnerabilities and weaknesses are checked in each tool?

The detection capabilities and diagnostic resolution of the individual tools differ significantly. In addition, their diagnostic messages lack a uniform and comparable form. This way, their comparison necessitates mapping the individual targeted sets of weaknesses and diagnostic messages into a consistent basis. This needed a white-box reverse engineering approach. Therefore, we performed a qualitative analysis that:

1. identifies the checking rules applied;
2. maps them into weaknesses or vulnerabilities;
3. estimates the classes of vulnerabilities remaining uncovered.

Checking rules have a slightly different meaning, specified as follows. Rules of Securify represent the checks of the violation of patterns. Securify2 provides a list of checked antipatterns. Slither's rules designate the different detectors that identify anomalies in the smart contracts. SmartCheck's checking rules are determined by evaluating all Xpaths that the tool can check; the Xptah represents a rule and a specific pattern and can be retrieved as output when it identifies anomalies. Mythril and Remix rules consider the output of each module of the SA. The number of rules of Oyente, Osiris, and HoneyBadger represents the number of different outcomes that they provide.

We use examples, descriptions, definitions, recommendations, exploits, and other sources linked to checking rules (into the documentation and sometimes into the tool's code) to identify the ones related to the vulnerabilities (the tools also identify other defects). Finally, we map the checking rules to vulnerabilities (and consequently to classes) [124].

The white-box analysis used all the program and documentation-related information sources to reverse engineer the checking rules and their relevance for vulnerability detection.

Table 9 highlights, for each tool, the resulting rules and the vulnerability-related ones, showing how many vulnerabilities the tool can identify (and the number of classes involved). A detailed mapping is available at [124].

Table 9: Analysis of the internal codes of the tools.

Tools	**Resulting rules**	**Vulnerabilities related rules**	**Vulnerabilities involved**	**CWE-IDs involved**
Securify	10	10	7	5
Securify2	43	25	18	7
Slither	71	25	17	8
SmartCheck	85	42	17	8
Remix-IDE	22	13	10	6
Mythril	17	15	13	7
Oyente	7	7	6	5
Osiris	10	7	6	4
HoneyBadger	9	5	5	3

Table 10: Anticipated and uncovered vulnerabilities.

Tools	**CWE-20**	**CWE-284**	**CWE-330**	**CWE-345**	**CWE-400**	**CWE-668**	**CWE-669**	**CWE-682**	**CWE-691**	**CWE-703**
Securify			x	x	x	x		x		
Securify2			x	x	x					
Slither				x	x					
SmartCheck	x		x							
Remix-IDE	x		x	x				x		
Mythril	x			x	x					
Oyente	x		x	x	x					x
Osiris	x	x	x	x	x					x
HoneyBadger	x	x	x	x	x	x				x

Table 10 shows instead the vulnerability classes escaping detection completely. An x in cell (z,y) means that the tool in row z can NOT find any vulnerability belonging to the class in column y. Analyzing the table, we can observe some facts:

- *Securify*, *Securify2*, and *Slither* are the only ones capable of detecting vulnerabilities in CWE-20 (*Improper Input Validation*).
- CWE-330 (*Use of Insufficiently Random Values*) can be detected only by *Mythril* and *Slither*.
- Only *SmartCheck* investigates CWE-345 (*Insufficient Verification of Data Authenticity*) and two tools (*SmartCheck* and *Remix*) CWE-400 (*Uncontrolled Resource Consumption*).

Table 10 highlights that no tool in the selection covers by design the entire fault model; thus, diagnostic coverage metrics need a finer resolution of the individual classes intended to be covered by it. We refer to this subset of the fault model described in Section 3.3.2 as the *working domain* of the particular tool.

Besides the view based on classes, we investigated the individual vulnerabilities and found a few that the entire set of SATs cannot detect.

Vulnerabilities out of the detection capabilities of our set of static analysis tools:

- *Missing Protection against Replay Attack* (CWE-345): it occasionally depends on a specific function of the ERC-20 token [125].

Vulnerabilities-related weaknesses related to specific constructs of Solidity that would permit to catch: *Malicious Libraries* (CWE-668), *Requirement Violation* (CWE-20), *Type Casts* (CWE-345), *Insufficient Gas Griefing* (CWE-691).

4.2 Experimental settings and methodology

This section focuses on the settings (e.g., data sets and subsequent creation of a reference ground truth) and the methodology of the experimental campaign.

4.2.1 *The Reference Dataset*

We built a reference dataset of smart contracts, extracting randomly smart contracts (around 400) from the Etherscan (the Ethereum blockchain explorer) [126].

A smart contract can contain different logic contracts (contract keyword) that include functions (function keyword). A measure of the logical contracts or functions of smart contracts helps to highlight their essential characteristics.

The Thesis uses the following definitions for the number of lines of software. *Physical lines of code* (LOC) are the total number of lines of a smart contract's code. *Logical lines of code* (LLOC) identify the number of lines of code that are neither comments nor empty. These measures indicate the length of a program; the programming style strongly influences them.

4.2.2 *The Pilot Set*

The lack of standard benchmarks to compare Solidity checking tools necessitates the creation of a pilot set, a set of representative smart contracts with known vulnerabilities as a ground truth comparison basis. For instance, without additional knowledge, we cannot distinguish between *true positives* (TP – correct detection of an existing vulnerability) and *false positives* (FP – false detection of a non-existing vulnerability) and quantify the vulnerability detection precision of the individual tools.

Thus, we extracted a subset of contracts from the reference dataset for manual inspection, which constitutes our pilot set to assemble a *ground truth* [124]. Smart contracts selected from the reference dataset into the pilot set had to fulfil the following constraints inspired by benchmarks principles.

- *Representativeness of typical domains*. At least one contract of the set must belong to each of the top 5 categories: gambling, exchange, games, finance, and properties [127].
- *Compliance*. All tools can process all contracts without errors.
- *Representativeness to Solidity*. Contracts must contain the main features of the language (e.g., functions that exchange Ether, assembly usage, low-level calls, libraries, structures, arrays).

- *Representativeness of the kind of vulnerabilities.* The whole set of contracts must contain all kinds of vulnerabilities predicted by SA tools on the reference dataset.

Although the number of contracts in the pilot set is limited to 15, the number of logical contracts is 88 (with a total LLOC of 4684), and it is comparable to the dataset used in other works [57]. The total number of existing vulnerabilities is 486; moreover, a single line may contain multiple vulnerabilities. Accordingly, we consider vulnerable rows instead of total vulnerabilities. After finding a vulnerability in a row, a manual inspection is required to apply a countermeasure; thus, the probability of finding other existing ones is very high. The number of vulnerable rows in the pilot set is 411.

Table 11 summarizes the main characteristics of the contracts in the pilot set.

Table 12 provides the total number of vulnerabilities (in the first row) and vulnerable rows (in the second one) per class. In the rest of the Thesis, the term vulnerability refers to the vulnerable row unless otherwise specified.

4.2.3 *Performance Indicator Definitions*

Vulnerability detection is a classification task; accordingly, we use the standard statistical metrics for binary classification to quantify the behaviour of the individual tools. The *confusion matrix* is a 2x2 matrix used to describe the classifier behaviour. Columns and rows represent the true values and classifier results. The matrix's main diagonal represents the correct classification in terms of *true positives* (TP) - correct detection of an existing vulnerability - and *true negatives* (TN) - correct assessment of no vulnerability -. The anti-diagonal matrix contains the false classification: *false negatives* (FN – missed detection of an existing vulnerability) and *false positives* (FP – false detection of a non-existing vulnerability).

Based on the confusion matrix several performance indicators can be defined [128].

Table 11: Pilot set characteristics.

Smart Contracts IDs	**Characteristics**				
	LOC	***LLOC***	***Logic Contracts***	***Functions***	***Vulnerable rows***
0x01a5c0	190	133	1	22	13
0x16eb29	891	298	9	54	22
0x1cdcc3	776	408	8	113	27
0x239669	244	121	4	24	24
0x4b89f8	929	650	8	41	27
0x5571d1	297	145	4	24	10
0x605cc9	640	185	4	37	14
0x607620	180	149	4	14	25
0x999999	488	252	1	18	42
0xaa4de9	709	269	3	37	22
0xbc205b	1267	616	7	85	27
0xbd3149	227	108	5	22	23
0xd82556	1031	561	8	53	45
0xe042c2	1299	521	12	80	42
0xfd77ef	564	268	5	43	48
Total	9732	4684	83	667	411

Table 12: Pilot set vulnerabilities grouped by CWE-IDs.

Analysis	**CWE-IDs**								
	CWE -20	***CWE -284***	***CWE -330***	***CWE -400***	***CWE -668***	***CWE -669***	***CWE -682***	***CWE -691***	***CWE -703***
Tot. Vuln. (486)	119	256	4	12	24	6	12	34	19
Vul. Rows (411)	74	235	4	12	24	4	11	29	18

Recall (R) is the percentage of detected anomalies (true positives) over all the anomalies (true positives + false negatives):

$$R = TP \mathbin{/} (TP + FN) \tag{1}$$

The term coverage is often used with the same meaning of recall. In the rest of the paper, we use the term coverage.

Precision (P) is the ratio of true and total fault indications penalizing false alerts:

$$P = TP \mathbin{/} (TP + FP) \tag{2}$$

F_1 score is the harmonic mean between precision and recall, defined as follows:

$$F_1 = 2 \times P \times R \mathbin{/} (P + R) \tag{3}$$

Accuracy (A) describes the ratio of correct diagnostic results among all:

$$A = (TP + TN) \mathbin{/} (TP + TN + FP + FN) \tag{4}$$

However, accuracy can be misleading when used for unbalanced data, like those we expect for the SA tool with dominating true negatives.

Balanced accuracy (BA) [129] normalizes true negatives and true positives prediction, as the average of the true positive and negative rates:

$$BA = (TP \mathbin{/} (TP + FN) + TN \mathbin{/} (FP + TN)) \mathbin{/} 2 \tag{5}$$

All the metrics above become in case of a perfect fault coverage with no false alerts to 1.0.

4.2.4 *Methodology*

We use the static analyzers of Section 4.1.1, and datasets of Sections 4.2.1 and 4.2.2.

Each tool processed the whole reference dataset. Checking a contract under test may result in a successful test run or a processing failure - an improper or incomplete test run with partial (or no) diagnostic outcome -. A successful test run may deliver a no-vulnerability found message (negative result) or a vulnerability indication complemented with diagnostic information on the location and type. At first, we focus only on the vulnerability detection capabilities and refer to all cases with a vulnerability indication as a positive.

We observed many successful runs and sporadic errors in processing smart contracts using our selected SA tools for the reference dataset. Processing errors happen mainly for two reasons:

- the tool uses an external module that exhausts memory resources (e.g., Securify2);
- the tool covers the Solidity language only incompletely.

The set of smart contracts that are processed by each tool without any errors forms the *reduced* dataset.

We consider the number of *lines of code* (LOC) of each contract and define the *location of detection* (LoD) as the line of a smart contract where a tool detects a positive. Each positive is detected by a vulnerability-related rule. Each vulnerability-related rule is mapped to a vulnerability and consequently to a CWE-ID of our taxonomy. The tuple (*tool*, *address*, *LoD*, *CWE-ID*) identifies a positive that a specific *tool* detects in a smart contract under test.

Determining whether each positive is a true or false positive requires a massive amount of (manual) work. Thus, we used the annotated pilot set (Section 4.2.2) extracted from the *reference dataset*. The ground truth of the pilot set permits calculating the upper-limit of class coverage for each tool and CWE-ID. Moreover, the ground truth permits determining whether each positive outcome is a TRUE positive or a FALSE one, and whether each negative is a TRUE negative or a FALSE one. We used a new tuple (*tool*, *address*, *LoD*, *CWE-ID*, *diagnosis*), adding the field diagnosis, which can assume values TP, FP, TN, and FN.

As shown previously in Section 4.1.2, no tool in the selection covers by design the entire set of vulnerabilities; thus, diagnostic coverage metrics need a finer resolution of the individual classes intended to be covered by it. We refer to this subset of the anticipated anomaly classes (described in Chapter 3) as the *working domain* of the particular tool. A detection is missing (for a specific tool) if a vulnerability from its

Table 13: The upper-limit of class coverage.

Tools	**CWE-20**	**CWE-284**	**CWE-300**	**CWE-400**	**CWE-668**	**CWE-669**	**CWE-682**	**CWE-691**	**CWE-703**
Securify	1.0	0.2	N/A	N/A	N/A	1.0	N/A	0.9	1.0
Securify2	1.0	1.0	N/A	N/A	0.6	1.0	0.1	1.0	1.0
Slither	0.6	0.8	1.0	N/A	0.6	1.0	0.1	0.9	1.0
SmartCheck	N/A	0.8	N/A	1.0	1.0	1.0	0.1	0.3	1.0
Remix-IDE	N/A	0.1	N/A	0.9	0.9	1.0	N/A	0.7	0.1
Mythril	N/A	0.1	1.0	N/A	0.6	1.0	1.0	0.7	1.0
Oyente	N/A	0.1	N/A	N/A	0.6	1.0	1.0	0.8	N/A
Osiris	N/A	N/A	N/A	N/A	0.6	1.0	1.0	0.8	N/A
HoneyBadger	N/A	N/A	N/A	N/A	N/A	1.0	1.0	0.2	N/A

working domain escapes detection (false negative). We use for this the pilot set, and the purpose of this experimental campaign is to assess how tools perform in their respective working domain, determining how good are built.

Finally, we provide a benchmark evaluating each SA tool as a black box.

4.3 Testing Performance

This section focuses on testing performance, starting from determining the upper-limit of class coverage (e.g., data sets and subsequent creation of a reference ground truth) and the methodology of the experimental campaign.

4.3.1 *The Upper-limit of Class Coverage*

Which is the upper-limit of class coverage for each tool?

Manual inspection allowed to identify vulnerabilities for each class. The classwise ratio between the number of vulnerabilities detectable by a particular tool and the ones present in the pilot set indicates the upper-limit of the coverage, assuming vulnerabilities are uniformly distributed (Table 13). A concrete example can help clarify values in the table. Consider Slither and the class CWE-284. In the pilot set, the number of existing vulnerabilities that belong to the class CWE-284 is 235. Among them, the occurrences of the vulnerability types that Slither can detect are 185. Thus, the upper bound of the CWE-284 coverage is 185/235, i.e., 0.8.

Class CWE-345 is omitted from the table because no vulnerability belongs to this class in the reference and pilot set. The following section investigates the quantification of the testing performance of the individual static analyzers.

Some observations follow:

- For each CWE-ID in the taxonomy, there is *at least one tool* for which the upper-limit of class coverage is 1.0;
- *All tools* have the upper-limit of CWE-669 coverage of 1.0;
- One tool that has the upper-limit of the coverage of 1.0 for CWE-284 (*Securify2*), CWE-400 (*SmartCheck*), CWE-668 (*SmartCheck*) and CWE-691 (*Securify2*).
- *Mythril* has the maximum upper-limit coverage of 1.0 for the three classes with fewer occurrences (CWE-330, CWE-669, CWE-682).

4.3.2 *How Tools Perform in the Working Domain*

How do tools perform in their working domain?

We focus on the pilot set. By combining the evaluation of tools in Section 4.1.2 and the findings described in Section 4.2.3, we can build confusion matrices for each class of the taxonomy.

Table 14 and Table 15 provide coverage and precision for each tool and class, respectively. Light-red cells (x, y) indicate that the tool in row x chose NOT to detect vulnerabilities of the class y. In both tables, light-green cells highlight the best tool x for each class y. HoneyBadger is omitted from the tables because it finds no TP in its working domain in the pilot set.

The following example shows how the values of Table 14 are calculated. The vulnerabilities in the pilot set belonging to the working domain of Slither in the CWE-284 class are 185 (the total number of vulnerabilities in the CWE-284 class is 235). Slither detects 170 of them; therefore, the coverage is the ratio of 170 to 185, i.e. 0.9.

Table 14: Classwise vulnerability coverage – tool working domain.

Tools	**CWE-20**	**CWE-284**	**CWE-330**	**CWE-400**	**CWE-668**	**CWE-669**	**CWE-682**	**CWE-691**	**CWE-703**
Securify2	0.5	0.9	N/A	N/A	0.0	0.8	0.1	0.2	1.0
Securify	0.2	0.6	N/A	N/A	N/A	0.0	N/A	0.2	0.4
Slither	0.2	0.9	0.0	N/A	0.5	0.8	0.0	1.0	0.7
SmartCheck	N/A	0.9	N/A	0.4	0.0	0.3	0.0	0.3	0.0
Remix-IDE	N/A	1.0	N/A	0.7	0.7	0.8	N/A	0.8	0.0
Mythril	N/A	0.8	1.0	N/A	0.3	1.0	0.0	0.1	0.4
Oyente	N/A	0.0	N/A	N/A	0.0	0.5	0.3	0.1	N/A
Osiris	N/A	N/A	N/A	N/A	0.0	0.3	0.4	0.1	N/A

Table 15: Precision of tools.

Tools	**CWE-20**	**CWE-284**	**CWE-330**	**CWE-400**	**CWE-668**	**CWE-669**	**CWE-682**	**CWE-691**	**CWE-703**
Securify2	0.5	0.8	N/A	N/A	0.0	1.0	0.3	0.3	0.2
Securify	0.6	0.3	N/A	N/A	0.0	0.0	N/A	0.5	0.8
Slither	1.0	1.0	0.0	N/A	0.9	1.0	0.0	0.7	1.0
SmartCheck	N/A	0.9	N/A	1.0	0.0	1.0	0.0	0.3	0.0
Remix-IDE	N/A	0.5	N/A	1.0	0.8	1.0	N/A	0.9	0.0
Mythril	N/A	0.8	1.0	N/A	0.8	0.4	0.0	0.5	0.9
Oyente	N/A	0.0	N/A	N/A	0.0	1.0	0.6	0.3	N/A
Osiris	N/A	N/A	N/A	N/A	0.0	1.0	0.7	1.0	N/A

We would emphasize that the different tool's precision values cannot be directly compared in this section because they refer to different sets of vulnerabilities (TP+TN).

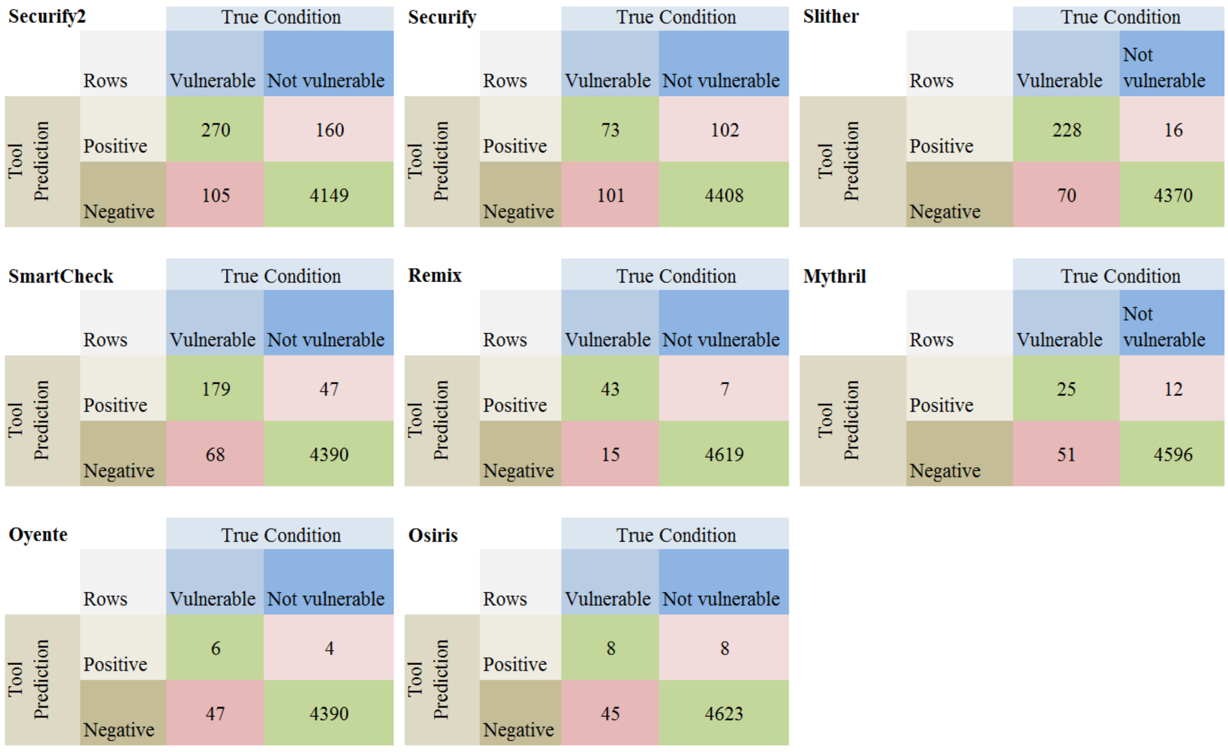
Through the tables, developers can determine how well their tools perform on the vulnerabilities they decided to target. Some observations follow:

- *Slither* has a high precision (>= 0.9) in five classes and has a high coverage (>= 0.9) in two.
- *Securify2* and *Slither* perform well both in precision and coverage in two classes (CWE-284 and CWE-669). *Securify2* covers the class CWE-703 completely (but with low precision) and has the highest coverage in CWE-20.
- *Mythril* has complete coverage and perfect precision in CWE-330.
- *Remix* has good coverage and high precision in CWE-669 and CWE-691.

Next, we want to assess the aggregate coverage and precision of the tools without distinguishing among classes. We generate the new confusion matrices (Figure 18) and summarize the main derived metrics in Table 16. Green and light green indicate the top two performances in the tools working domain. Some observations follow:

- The working domains of *Securify2* and *Slither* contain the largest and second-largest number of vulnerabilities, respectively.
- *Slither* and *Remix* have the best and the second-best performance both in coverage and precision. However, Remix targets fewer vulnerabilities than Slither;
- *Securify2* covers more than 70% of vulnerabilities, with low precision (0.6);
- *Mythril* has low coverage (0.3) and a precision lower than 0.7.

In this section, we analyzed the behaviour of the tools in their working domain. Once we identified the tool's working domains, we quantified their classwise and aggregate detection capabilities (Tables 14-16).



Securify2		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	270	160
	Negative	105	4149

Securify		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	73	102
	Negative	101	4408

Slither		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	228	16
	Negative	70	4370

SmartCheck		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	179	47
	Negative	68	4390

Remix		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	43	7
	Negative	15	4619

Mythril		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	25	12
	Negative	51	4596

Oyente		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	6	4
	Negative	47	4390

Osiris		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	8	8
	Negative	45	4623

Figure 18: Confusion matrices in the tool working domain.

Table 16: Tools performances – working domain.

Pilot Set	**Sfy2**	**Sfy**	**Sli**	**SmC**	**Rmx**	**Myt**	**Oye**	**Osi**
Precision	0.63	0.42	0.93	0.79	0.86	0.68	0.60	0.50
Coverage	0.72	0.42	0.77	0.72	0.74	0.33	0.11	0.15
Balanced accuracy	0.84	0.70	0.88	0.86	0.87	0.66	0.56	0.57
F_1 Score	0.67	0.42	0.84	0.76	0.80	0.44	0.19	0.23

Table 17: Classwise vulnerability coverage – tool benchmarking.

Tools	**CWE-20**	**CWE-284**	**CWE-330**	**CWE-400**	**CWE-668**	**CWE-669**	**CWE-682**	**CWE-691**	**CWE-703**
Securify2	0.4	0.9	N/A	N/A	0.0	0.7	0.1	0.2	1.0
Securify	0.2	0.2	N/A	N/A	N/A	0.0	N/A	0.2	0.4
Slither	0.1	0.7	0.0	N/A	0.3	0.7	0.0	0.9	0.7
SmartCheck	N/A	0.7	N/A	0.4	0.0	0.2	0.0	0.1	0.0
Remix-IDE	N/A	0.1	N/A	0.7	0.6	0.7	N/A	0.1	0.1
Mythril	N/A	0.1	1.0	N/A	0.2	0.5	0.0	0.1	0.3
Oyente	N/A	0.0	N/A	N/A	0.0	0.5	0.3	0.1	N/A
Osiris	N/A	N/A	N/A	N/A	0.0	0.2	0.4	0.1	N/A

4.3.3 *Tools Benchmarking*

What is the tool benchmark in the general case?

This section extends our benchmark on the capabilities of the individual tools in the general case evaluating the outcomes obtained by processing the entire pilot set. We use the same four performance metrics as above. The number of TPs and FPs per tool and class, considering the tool working domain and tools as black boxes, coincides. Instead, FNs and TNs differ. Accordingly, because of the definitions (1), (2), (3), (5), the precision of Section 4.3.2 (Table 15) and this section coincide; conversely, the other metrics differ.

Table 17 provides the coverage for each tool and class. These results can help choose the best tool once specific vulnerabilities are targeted. Some observations follow:

- *Securify2* has the highest coverage in classes CWE-20, CWE-284, and CWE-703 (with low precision).
- *Slither* and *Remix* have the highest coverage, respectively, in classes CWE-669 and CWE-691 (Slither), and CWE-400 and CWE-668 (Remix) - with high precision-.

Securify2		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	270	160
	Negative	141	4113

Securify		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	73	102
	Negative	338	4171

Slither		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	228	16
	Negative	183	4257

SmartCheck		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	179	47
	Negative	232	4226

Remix		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	43	7
	Negative	368	4266

Mythril		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	25	12
	Negative	386	4261

Oyente		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	6	4
	Negative	405	4269

Osiris		True Condition	
	Rows	Vulnerable	Not vulnerable
Tool Prediction	Positive	8	8
	Negative	403	4265

Figure 19: Confusion matrices for benchmarking.

Table 18: Tools performances – benchmark.

Pilot Set	Sfy2	Sfy	Sli	SmC	Rmx	Myt	Oye	Osi
Precision	0.63	0.42	0.93	0.79	0.86	0.68	0.60	0.50
Coverage	0.66	0.18	0.55	0.44	0.10	0.06	0.01	0.02
Balanced accuracy	0.81	0.58	0.78	0.71	0.55	0.53	0.51	0.51
F1 Score	0.64	0.25	0.70	0.56	0.19	0.11	0.03	0.04

- *Mythril* and *Osiris* have the highest coverage in the class CWE-330 and CWE-682.

Figure 19 shows the new confusion matrices by aggregating data without distinguishing among classes. Table 18 provides the overall benchmark results. Again, green and light-green cells highlight the best and second-best results. Some observations follow:

- *No tool* can cover more than 66% of the vulnerabilities in the pilot set (coverage), and no tool exceeds the value of 70% (*Slither*) in the F_1 score.

- *Securify2* has the best coverage (even if it misses 1/3 of vulnerable rows) and the second F_1 score performance. It has the highest balanced accuracy; however, it has low precision.

- *Slither* has a coverage of 0.10 lower than *Securify2*, but it has a very high precision (0.93). Moreover, it has the second performance (0.78) of balanced accuracy.

- *SmartCheck* has lower performances than *Slither* by about 0.10 in every metric.

Dedicated tools with a specific purpose and limited working domain have worst performances than others when analyzing a broad set of vulnerabilities.

This evidence permits us to argue that considering the coverage alone (disregarding the false positives cost), *Securify2* is a good choice. If we target a balance between precision and coverage, *Slither* is to be preferred. However, the low diagnostic accuracy (coverage) demands an investigation on the potential improvement using a combination of tools.

4.4 Validity and Limitations

The quality of smart contracts has a significant influence on our analysis results. For instance, widely used quality estimators, like *COnstructive QUALity MOdel* (COQUALMO [130]), use many influencing factors. The sample set of smart contracts considered in our study originated from a public repository with no information on such essential factors as the developers' skills. Qualified people in software security-oriented companies possessing a sophisticated background could produce better contracts regarding software weaknesses and vulnerabilities than the cuff developers. Variance in skills influences both the total number and distribution of the vulnerabilities in smart contracts.

The uncovered classes and vulnerabilities of Table 10 are independent of the smart contract selection.

We suppose that eliminating over-represented contracts makes the distribution of CWE classes in a general dataset and pilot set comparable. Tools have different precision and coverage in each class (as highlighted in Tables 15 and 17). Thus, precision and coverage in a set of smart contracts depend on the distribution of CWE classes. We want to compare the CWE distribution of the pilot set and a general set. The pilot set is processed by each tool (each tool contributes to the CWE distribution) without any processing error. In our set, smart contracts from the reference dataset, which are processed without any errors by each tool, form the *reduced dataset* (around 300 smart contracts). The distribution of the positive rows found by the tools and grouped into CWE classes of a reduced dataset is comparable to distributions in the pilot set (Figure 20). Then, with these premises, findings on precision and coverage are generalizable.

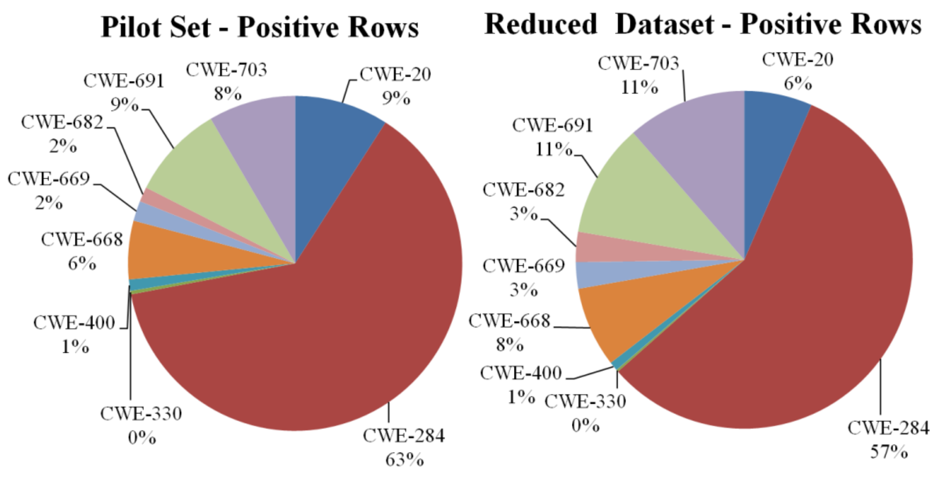


Figure 20: Distribution of positives in the pilot and reduced dataset.

5 IMPROVEMENT OF THE SMART CONTRACT SECURITY

Reducing the number of residual vulnerabilities is crucial and can be achieved through static analysis. In Chapter 4, we analyzed how the security of smart contracts can be evaluated by using static analyzers at the early stage of the software life cycle. This section tackles the third research question we address: *How to improve the smart contract security using SA tools?* (**RQ3**)

We address the problem by considering the outcomes of SA tools. Increasing the TPs means increasing the detected vulnerabilities and the number of FPs (costly in terms of time and resources spent for their reduction - Section 2.4.3). The first goal is to increase the number of TPs without increasing FPs excessively. Moreover, increasing TPs means decreasing FNs. However, FNs cannot be eliminated and are dangerous because they enable subsequent successful attacks. To increase the smart contract security, we have to look for which FNs are most critical to prioritise mitigation actions. Moreover, each TP is mapped to a CWE-ID in our taxonomy and is also identified by its physical location in a contract. Identifying clusters of TPs (characterised by belonging to a specific CWE-ID) in specific areas of contracts allows for identifying similar clusters of vulnerabilities in the same locations. This information suggests specific areas in Solidity smart contracts where the search for vulnerabilities should focus.

Thus, Section 5.1 treats the improvement of the coverage by combining tools. Sections 5.2 and 5.3 respectively address the prioritization of vulnerabilities escaping detection and where vulnerabilities are more likely located. Finally, Section 5.4 highlights some remarks.

5.1 Coverage Improvement

Is it possible to improve the coverage by using a combination of tools? As the previous experiments indicated highly different classwise detection capabilities of the individual tools, we may consider combining them.

There are several ways to combine tools [131]. We use multiple SA tools to test a smart contract from the pilot set in our experiment and consider the combined test result *positive* if any of the tools have a positive outcome (OR type combination). This way, fault coverage improves, at a price to increase the number of false positives.

True positives are rows in which at least one of the tool predictions is positive, and the row really contains at least one vulnerability. *False positives* are rows in which at least one of the tool predictions is positive but they contain no vulnerabilities. *False negatives* and *true negatives* follow accordingly.

We run our experiments for all the possible combinations of 2, 3 or 4 tools and report in Table 19 the combinations with the highest fault coverage.

More *false positives* to be analyzed (i.e., the precision of the combination is less than that of the best tool) is the cost paid for using the combination. As an asymptotic result, we report that using all the considered tools together 382 of the 411 vulnerable lines are detected, with the coverage of 0.93. The best coverage combining two tools

Table 19: Tool combinations.

Pilot Set	**Sfy2 Sli**	**Sfy2 SmC**	**Sfy2 Rmx**	**Sfy2 Myt**	**Sfy2 Sli SmC**	**Sfy2 Sli Rmx**	**Sfy2 Sli SmC Rmx**	**Sfy2 Sli SmC Myt**
Precision	0.67	0.62	0.65	0.64	0.64	0.68	0.64	0.64
Coverage	0.81	0.76	0.71	0.68	0.88	0.85	0.90	0.90
Balanced accuracy	0.89	0.86	0.84	0.82	0.92	0.91	0.93	0.93
F1 Score	0.73	0.68	0.68	0.66	0.74	0.75	0.75	0.75

remains at 0.81. By combining three tools, the coverage reaches at best the value of 0.88 (*Securify2 – Slither – SmartCheck*). The second-best solution (*Securify2 – Slither – Mythril*) has a value of 0.85 with no meaningful difference in *balanced accuracy* and *F_1 score*. Moreover, we investigated four-tool combinations. We obtained the best result by adding *Remix* or *Mythril* to the best combination of three tools: the coverage and precision have a value of 0.90 and 0.64.

Another evaluation targeted the coverage for each of the classes rather than considering the aggregate. Proper combinations should include: a tool between *Securify2*, *Securify*, and *Slither* for CWE-20; *Mythril* to cover CWE-330; *SmartCheck* or *Remix* for CWE-400. The best combination contains four tools (*Securify2 – Slither – SmartCheck - Mythril*). This combination is also one of the best four-tool combinations found earlier, which appears thus to be the best from the point of view of both aggregate and classwise vulnerability coverage.

Finally, we report the cost in terms of execution run time. The architecture used is a Virtual Machine (Ubuntu64 based, 8GB of memory) that ran in a Linux Server with 24 GB. Slither, Remix, and SmartCheck require less than a minute to analyze the entire pilot set. Securify2, Securify, and Mythril need 30 minutes, 2 hours, and 4 hours respectively.

We run some experiments over big contracts (between 1500 and 6300 LOC). The maximum required time for the analysis of a big smart contract remains within a minute for *Slither*, *Remix*, and *SmartCheck*. *Securify2*, *Securify*, and *Mythril* require a maximum time of one hour, three hours, and seventeen hours.

5.2 Vulnerabilities Prioritization

Even using combinations of tools we have undetected vulnerabilities. We want to investigate whether they are all equally critical or if some can be more dangerous.

To do this, we need to determine their severity, thus identifying the most critical types of undetected vulnerabilities, which would become a top priority in an effort for mitigation. We investigated several prioritization methods (e.g., [132], [133], [134]) and choose two ([133], [134]) that are dealing explicitly with vulnerability prioritization: the typical severity of the *Common Attack Pattern Enumeration and Classification* (CAPEC); ii) the severity determined by the *Common Weakness Scoring System* (CWSS) developed by the CWE.

5.2.1 *CAPEC*

5.2.1.1 Overview

CAPEC [133] provides a public catalog of common attack patterns that an attacker uses to exploit known weaknesses. Each pattern captures the design and execution of an attack, thus providing information about the severity and mitigation of the attack. CAPEC uses four levels of abstraction for attack patterns (ordered by increasing level of detail): *category*, *meta*, *standard*, *detailed*.
Currently[3], the CAPEC archive contains 546 attack patterns, divided into two main hierarchical views:

- *Mechanisms of attack*, consisting of 9 categories, representing the fundamental mechanisms used to exploit a vulnerability;
- *Domains of attack*, represented by six different categories: *software*, *hardware*, *communications*, *supply chain*, *social engineering*, *physical security*.

CAPEC attack patterns capture the exploitation of weaknesses. Each attack pattern contains some information, including a description of the attack, relationships to other attack patterns, and a *typical severity* (which we will use for prioritization).

5.2.1.2 Mapping CAPEC patterns to vulnerabilities

We consider *vuln* as a type of vulnerability that belongs to our taxonomy. To determine the mapping between vulnerabilities and CAPEC attack patterns, we used the following method:

∀ *vuln*

4. Considering the known exploit scenarios:
    a. we determined the *mechanism of the attack*;
    b. we extracted some *keywords*;
5. We identified a set of CAPEC patterns (*C_set*) by performing a *keyword* search on the CAPEC list, filtered through the *mechanism of the attack*.

[3] In December 2021, the latest version of the CAPEC list is 3.6.

6. Then, we selected the maximum severity of attack patterns belonging to *C_set*.

5.2.2 *CWSS*

5.2.2.1 *Overview*

CWSS [134] provides a way to prioritize weaknesses by proposing a methodology combining three groups of metrics: base finding metric (information extracted from the weakness class), attack surface metric (barriers an attacker must overcome), and environmental metric (characteristics of the environment of the weakness). Each type of metric group is composed of several factors that, combined with appropriate weights, determine the metric's subscore. The combination of subscores determines the final CWSS score [134]. CWSS explicitly supports cases of incomplete information (factors taking *unknown* value) and allows ignoring irrelevant factors in the analyzed context (*not applicable* value).

5.2.2.2 *CWSS calculation*

For the scoring calculation, we followed [134]. Our basic assumption is that in case of a vulnerability, an attacker is always able to discover and exploit it. The main factors to consider are:

- *Common consequences* of the CWE-ID (associated with the vulnerability-related weakness) leading to potential *technical impacts* on the Blockchain;
- *Worst-case scenarios* in terms of *business impact*;
- *Vulnerability mitigation capability* provided by an internal (e.g., mandatory software construct) or external control (e.g., EVM).

The resulting score is a value from 0 to 100, then reported on a scale from 0 to 10.

5.2.3 *CAPEC and CWSS Severity*

CAPEC uses natively four severity classes (*critical*, *high*, *medium*, *low*), CWSS returns a score from 0 to 10. To compare the two scoring systems, we determine the CWSS severity class using the following score-to-severity conversion (used by NIST [135]): 0 - 3.9 *low*; 4.0 - 6.9 *medium*; 7.0 - 8.9 *high*; 9.0 - 10 *critical*.

Table 20 (using the acronyms in Table 7) shows the severity of classes and vulnerabilities of our taxonomy. The following colours indicate the severity: blue - *critical*, red - *high*, green – *medium*; grey represents vulnerabilities whose severity is *not identified*. The maximum severity of the vulnerabilities that belong to a specific class determines the severity of the class.

5.2.4 *Prioritization of False Negatives*

Once we have determined the severity, we prioritize vulnerability escaping detection (FNs). We process the manually annotated pilot set (Section 4.2.2) with the n-tool combinations of Section 5.1; the number of false negative types of each combination, grouped by severity, is shown in Figure 21 (CAPEC) and Figure 22 (CWSS). As an example, we examine the case of using a 3 (or more) SA tool combination. This setup results in fewer types of false negatives than a 2-tool combination.

By considering the union of the FN types in each scoring method, we can argue that:

- CWSS and CAPEC identify 6 FNs with *critical* or *high* priority (BU, TD, FE, IOU, Re, SA);
- CWSS and CAPEC identify 5 FNs with *medium* priority (CPL, ELT, GR, SF, VEF);
- CWSS and CAPEC identify 0 FNs with *low* priority;
- The two methods differ in identifying FNs with critical and high priority.

In general, technical and business impacts dominate the CWSS score. The values of these factors reflect the particular nature of the Blockchain and the resulting criticality that each breach entails. CAPEC cannot account for this specificity: the severity is a consequence once the pattern is determined. To summarize, CWSS focuses on consequences and CAPEC on the attack method.

Table 20: Severity of vulnerabilities and classes based on CAPEC and CWSS.

CWE-IDs	**CWE-20**	**CWE-284**	**CWE-330**	**CWE-345**	**CWE-400**	**CWE-668**	**CWE-669**	**CWE-682**	**CWE-691**	**CWE-703**
CAPEC: severity of vulnerabilities	**SA RV ELT**	**Atx UEW Usd VEF**	**GR**	**MPRA SM Ty**	**CPL Gs**	**BU TD ML SF**	**CU DUC EC**	**IOU AP**	**RLO FE Re TOD UEB IGG AJ**	**ED Us UV**
CAPEC: CWE-ID criticality	**H**	**C**	**M**	**H**	**M**	**C**	**C**	**H**	**C**	**M**
CWSS: severity of vulnerabilities	**SA RV ELT**	**Atx UEW Usd VEF**	**GR**	**MPRA SM Ty**	**CPL Gs**	**BU TD ML SF**	**CU DUC EC**	**IOU AP**	**RLO FE Re TOD UEB IGG AJ**	**ED Us UV**
CWSS: CWE-ID criticality	**H**	**C**	**M**	**C**	**M**	**C**	**C**	**C**	**C**	**H**

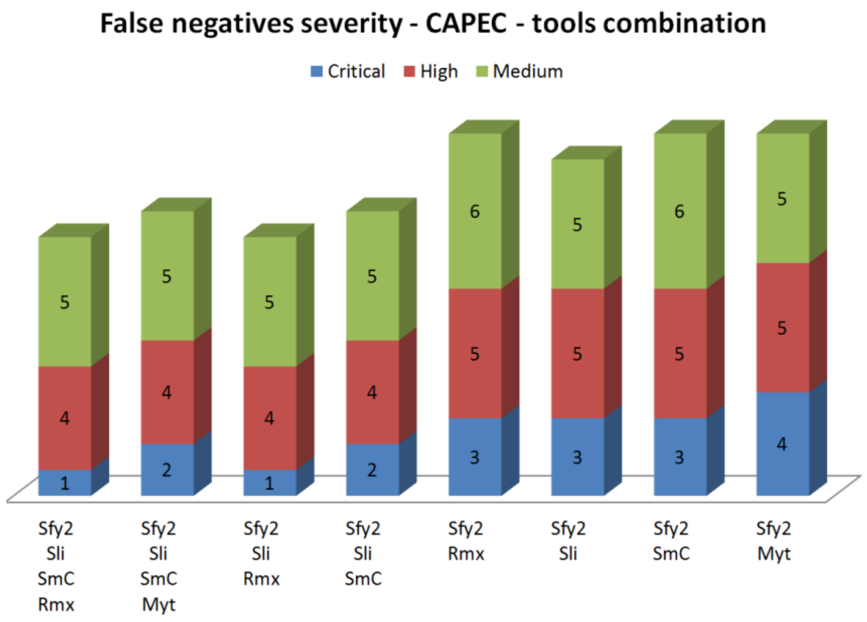


Figure 21: Types of false negatives (CAPEC), n-tool combinations.

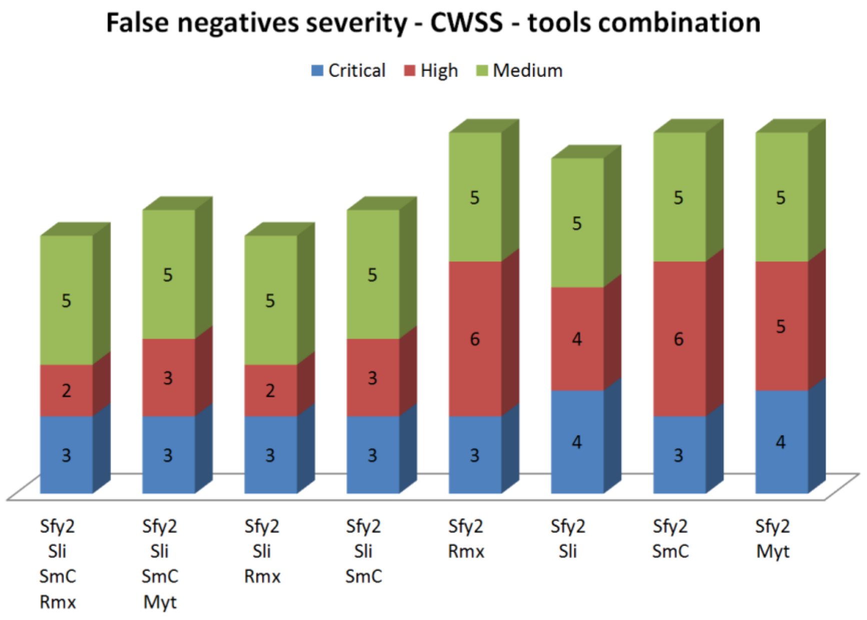


Figure 22: Types of false negatives (CWSS), n-tool combinations.

By analyzing the two figures, we can refine considerations in Section 5.1:

- the *Sfy2-Sli-SmC-Myt* four-tool combination, although it has fewer occurrences of FNs than *Sfy2-Sli-Rmx*, contains more critical (CAPEC) or high (CWSS) severity types.
- *Sfy2-Sli-SmC-Myt* and *Sfy2-Sli-SmC-Rmx* are equivalent in terms of coverage and precision; however, the combination that includes Remix contains fewer types of FNs with high or critical severity.

5.3 Location of Vulnerabilities

5.3.1 *Motivations*

Although vulnerabilities and analysis tools are a widely debated topic, the characterization of the position of vulnerabilities in Solidity smart contracts is surprisingly less investigated compared to other programming languages. Locating a specific class of vulnerabilities into smart contracts can help in different ways. On one side, tool developers can be guided for improving the vulnerability detection capabilities of the tool, while on the other side, software developers can produce more secure contracts focusing on the specific areas where such vulnerabilities are more likely located.
To locate vulnerabilities, we use the fault model proposed in Chapter 3. The starting point was the datasets of Section 4.2. We focus on vulnerabilities instead of vulnerable rows. We analyze the distributions of the locations where tools find positive outcomes. We create the ground truth of vulnerabilities for a subset S of smart contracts through manual inspection, and we first perform a comparison of the distributions within this set. Then we generalize our findings by comparing the distributions between the manually inspected subset and the full set. Such comparison allows us to identify where certain classes of vulnerabilities are located, suggesting specific areas in Solidity smart contracts where the search for vulnerabilities should focus.

5.3.2 *Experimental Settings and Methodology*

From the list of static analyzers defined in Section 4.1.1, we exclude HoneyBadger because of its low effectiveness.

We built an *experimental dataset* (more than 300 smart contracts [124]) starting from the reference dataset of Section 4.2.1, excluding contracts that generated some processing failures when processed by tools.

Each tool processed the whole experimental dataset. Testing each row of a contract can result in two different outcomes: *negative or positive detection*. By focusing on positives, each tool delivers results with its own codes and specific format. For each tool, we identified the codes related to the vulnerabilities belonging to our taxonomy, excluding the other codes from the analysis. Finally, we harmonized the results as described below.

Let consider the number of *lines of code* (LOC) of each contract and define the *location of detection* (LoD) as the line of a smart contract where a tool detects a positive. We define the *relative location of detection* (RLoD) as the ratio between the LoD and the LOC of the smart contract under analysis. The RLoD identifies where positives are located in the contracts. Thus, a tuple (*tool, address, RLoD, category*) represents a positive. *Tool* is the tool that identifies the positive, *address* is the smart contract address, *RLoD* is the relative location of detection, and *category* is the class of the CWE taxonomy the vulnerability belongs to.

Determining whether each positive is a true or false positive requires a massive amount of (manual) work. Thus, we extracted a subset composed of 15 contracts (referring to it as a pilot set) from the *experimental dataset*. Guided by the construction criteria of Section 4.2.2, we could use the same pilot set of that section.

We first analyzed the pilot set. The manual inspection permitted to determine the ground truth, i.e., to determine for each positive finding whether it is a TRUE positive or a FALSE one. In addition, it permitted to determine for each negative whether it is a TRUE negative or a FALSE one. For our purpose, however, we sought for the location of positives. We used the RLoD (defined above) to determine the location. We used a new tuple (*tool, address, RLoD, category, diagnosis*), adding the field diagnosis, which can assume values TP or FP. For each class of the taxonomy, we determined the location of the vulnerabilities by analyzing the TPs. Next, we analyzed our results by comparing the TPs to all the positives (including FPs). Then, by comparing the distributions of positives between the pilot and reference set and having a clue on the true positives, we tried to understand and discuss the generality of our findings.

Table 21: Pilot set analysis –vulnerabilities.

GT=486	**Securify2**	**Securify**	**Slither**	**SmartCheck**	**Remix**	**Mythril**	**Oyente**	**Osiris**
TP	320	40	229	182	43	27	6	8
FP	241	88	16	50	7	18	4	8
Coverage	0.67	0.08	0.48	0.38	0.09	0.06	0.01	0.02

5.3.3 *Pilot Set Analysis and Datasets Comparison*

As a preliminary step, the focus is on the sum of the occurrences of positives that the whole set of SA tools finds in the contracts. We found that 65% of the contracts in the reference set (55% in the pilot set) contain from 0 to 50 positives; 20% of the contracts in the reference set (33% in the pilot set) contain from 51 to 100 positives; the remaining 15% of the contracts in the reference set have more than 100 positives (12% in the pilot set).

Then we focus on the pilot set. First, we determine the *ground truth* (GT) for each class of our taxonomy. Then, we calculate the *coverage* as the percentage of detected vulnerabilities (TP) over all the vulnerabilities (GT).

The whole number of vulnerabilities (GT = 486), TPs, FPs, and the coverage are shown in Table 21. The analysis of the pilot set, detailed for each class, is shown in Table 22. The whole set of SA tools identifies 446 TPs over 486 vulnerabilities (92%). Results slightly differ from Chapter 4, as the focus is on vulnerabilities instead of vulnerable rows.

As observed, the tool coverage change based on the class; thus, the coverage of a dataset depends on the class distribution. Figure 23.a highlights the distributions of CWE categories in the pilot (red) and reference set (blue). As it can be observed, the classes have comparable frequency distributions in the two sets.

Moreover, Figure 23.b shows that CWE-345 (in grey) has no positives in either set; thus, we decided to exclude this class from further analysis.

Table 22: Coverage detailed for each class.

	CWE-20	CWE-284	CWE-330	CWE-400	CWE-668	CWE-669	CWE-682	CWE-691	CWE-703
Securify2	0.7	0.8	-	-	-	0.5	0.1	0.2	1.0
Securify	0.1	0.1	-	-	-	-	-	0.2	0.4
Slither	0.1	0.7	-	-	0.3	0.5	-	0.7	0.6
SmartCheck	-	0.7	-	0.4	-	0.2	-	0.1	-
Remix	-	-	-	0.7	0.6	0.5	-	0.5	-
Mythril	-	0.1	1.0	-	0.2	1.0	-	0.1	0.4
Oyente	-	-	-	-	-	0.3	0.3	0.1	-
Osiris	-	-	-	-	-	0.3	0.3	0.1	-
SA toolset	0.8	1.0	1.0	0.8	0.8	1.0	0.8	1.0	1.0

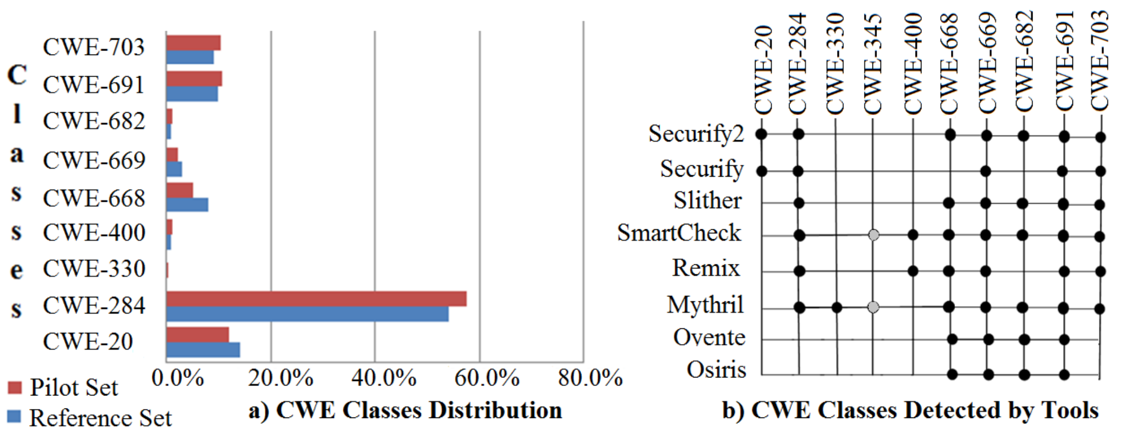


Figure 23: Data overview – vulnerability location. On the left, the distribution of CWE classes. On the right, which classes tools can detect.

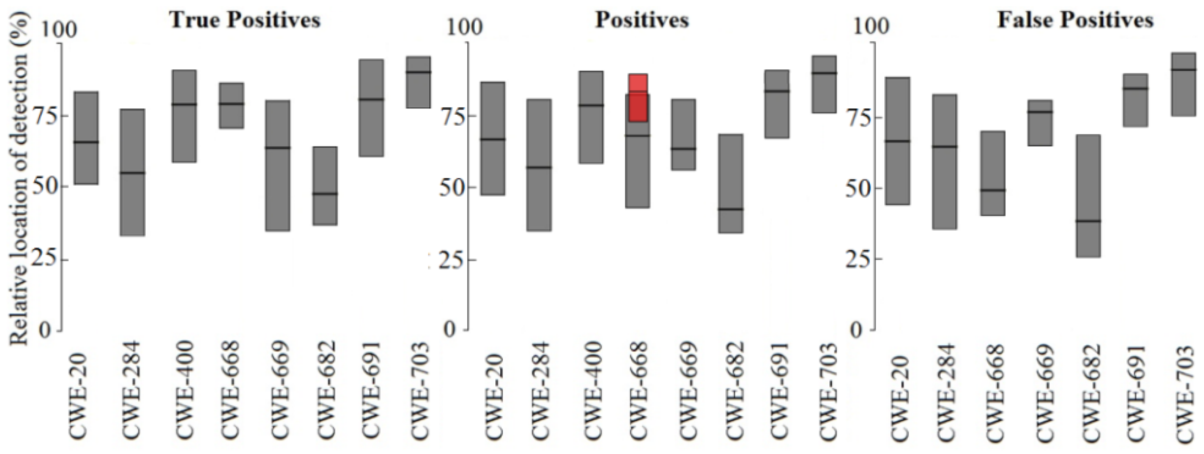


Figure 24: Comparison of RLoD distributions in the pilot set.

5.3.4 *Analysis of Vulnerability Location*

This section analyzes the location of specific classes of vulnerabilities. To perform a comparison of distributions, we use boxplots with the median and the *interquartile range* (IQR), the interval between the *upper hinge* (UH – 75th percentile of the distribution) and the *lower hinge* (LH – 25th percentile).

First, we focus on the pilot set, in which we analyze locations of the true positives (TPs) and compare them with all positives. Then, whether the distributions are different, we look at the false positives (FPs). Finally, by comparing the distributions of positives between the pilot and reference set and having a clue on the true positives, we try to understand and discuss the generality of our findings.

5.3.4.1 *Comparison in the pilot set*

The distributions of the positives in the pilot set are shown in Figure 24. Boxplots compare distributions of true positives (on the left), positives (in the middle), and false positives (on the right). Each boxplot shows the interquartile range for a specific CWE-ID of the taxonomy. A horizontal line within the box represents the median. The red boxplot highlights the distribution of positives without SmartCheck and Osiris for the CWE-668. As only four positives for CWE-330 have been detected, we decided to exclude this class from further analysis.

When we consider true positives, we can observe that:

- CWE-20, CWE-400, CWE-691, CWE-668, and CWE-703 have an LH greater than 50%; thus, more than 75% of TPs locates in the second half of contracts. In particular, more than 75% of TPs locates in the last quarter of contracts for CWE-703;
- The CWE-682 IQR (50% of TPs) is between 40 and 65;
- The IQR of CWE-284 and CWE-669 spread between 35 and 77.

When comparing the TPs to the positives, we determine that distributions of CWE-20, CWE-284, CWE-400, CWE-691, CWE-682, and CWE-703 differ in the upper, lower hinge, and median less than 5%. As a result, the distributions of positives are representative of distributions of TPs.

CWE-668 distributions differ significantly in the LH. FPs (at the right of Figure 24) affect the distribution of positives. The manual inspection of the pilot set shows that Osiris and SmartCheck detected 36 FPs and no TPs. Thus, we analyzed the distribution of positives without the outcomes of SmartCheck and Osiris (P_{SCO}) (as shown in the red boxplot of Figure 24). TPs and P_{SCO} distributions do not differ significantly; therefore, locations of P_{SCO} are representative of locations of true positives.

Finally, the focus is on CWE-669. Again, the distributions differ in the LH. However, for values greater than the median, which has a value of 65, the distributions are comparable. Locations of positives represent locations of true positives over the median.

5.3.4.2 Generalization

Comparing the distributions of positives between the pilot and reference set and having a clue on the true positives permits us to discuss the generality of our findings.

Figure 25 highlights the comparison. Boxplots compare distributions between the positives of the pilot set (on the left) and the reference set (on the right). The red boxplot highlights the distribution of positives without the outcomes of SmartCheck and Osiris (CWE-668).

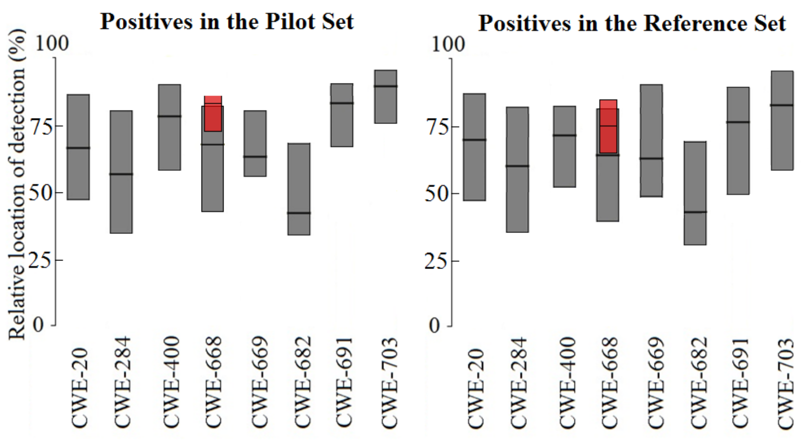


Figure 25: Comparison of location distribution between the pilot and reference set.

The distributions of CWE-20, CWE-284, and CWE-682 differ in the upper, lower hinge, and median less than 5%. Consequently, locations of positives in the reference set are representative of locations of positives in the pilot set. By considering the pilot set, locations of positives are representative of the locations of TPs. We can then generalize our findings:

- concerning CWE-20, 75% of vulnerabilities locates in the second half of the contracts;
- CWE-682 vulnerabilities locate mainly on the second third of contracts;
- CWE-284 spreads throughout all the length of contracts.

We analyze the distributions of positives of CWE-669 without the outcomes of SmartCheck and Osiris (Figure 26 – red boxplots). Distributions differ less than 10% in the UH median and LH, whereas the IQRs differ less than 5%. By generalizing findings from the previous section, 75% of vulnerabilities are located over the 60 (LH) value.

The distribution of CWE-400 in the reference set has lower values than the pilot set for the LH, median, and UH; however, the difference between the values is less than 10%. Therefore, the IQR between the distributions does not differ significantly: we can argue that distributions are comparable. We generalize findings with the same arguments used for CWE-669: 75% of vulnerabilities are located over the value of 50 (i.e., in the second-half of contracts).

CWE-691 and CWE-703 differ less than 5% in the UH and median but differ significantly in the LH. So, only the upper part of the distributions of positives (over the 50th percentile) is representative of TPs. This evidence allows us to locate the 50% of vulnerabilities in the last quarter of contracts for both distributions (median has a value greater than 75). No assumptions can be made for the lower part.

CWE-669 distributions have the same median. Analyzing the pilot set, we found that the distribution of positives represents TPs above the median. Thus, 50% of the vulnerabilities locates over the value of 65 (median of the distribution in the reference set).

5.4 Remarks

We can repeat the same considerations about the quality of the contracts used for our analysis as in Section 4.4. Different tools find positives in different classes. The difference has two root causes:

- tools aim to detect different vulnerabilities;
- tools can detect vulnerabilities with different capabilities.

There are several ways to combine tools whereby different choices lead to different results. As shown, using OR in the decision function increases TPs and FPs while using AND decreases TPs and FPs. Clearly, more complex decision functions could be used to pursue different objectives. The choice made here fits to improve the coverage. Tools detecting a vulnerability in a class can miss detecting other kinds of vulnerabilities belonging to the same class.

The severity of vulnerabilities and classes, determined through CAPEC and CWSS, is independent of the chosen set of smart contracts. The prioritization of FNs is qualitative rather than quantitative; this makes our results generically applicable to types that belong to our taxonomy.

True positives are a subset of vulnerabilities. As a general statement, static analysis tools can miss detecting vulnerabilities: this leads to wrong locate the range of TPs in case of multiple missed detections. Using eight tools reduces this risk (processing the pilot set with all tools permits finding 92% of the vulnerabilities).

Our preliminary findings are based on a pilot set. However, we generalize results using the distributions of positives in the reference set only in the case of comparable distributions.

Consider two distributions. This work uses the lower hinge (LH), median, upper hinge (UH), and interquartile range (IQR) for comparison. We consider that two distributions are comparable in the following two cases: i) the differences between LH, median, and UH values are less than 5%; ii) the difference is less than 10%, and the IQR ranges differ less than 5%.

To determine where vulnerabilities are most likely to be located, we use the relative location of detection (RLoD). This way, we can refer to the various areas of the smart contract without being bound by its length. For example, a range between 0 to 25 identifies the first quarter of a contract, a range from 26 to 50 identifies the second quarter. Other values follow accordingly.

6 CONCLUSIONS

Blockchain technologies promise an out-of-the-box solution to improve the security of distributed systems. The diffusion of smart contracts (automated execution of computerized transactions) allowed the development of applications in different areas (e.g., financial, medical, insurance, gaming, betting). Blockchain ensures data immutability, integrity and non-repudiability, protecting smart contracts, data, and transaction logs by a strong hash encoding. Design and coding faults in the smart contracts that implement a particular application can result in exploitable weaknesses. This problem is even more critical, considering that developers cannot patch smart contracts once deployed on the Blockchain.

Ethereum is one of the most widely used platforms for smart contract development and offers Solidity as its primary and Turing-complete programming language. There exist checking tools for smart contracts written in Solidity; however, they are immature due to the novelty and rapid evolution of the Solidity technology and a lack of a large set of vulnerability records. On the other hand, Solidity is just another new programming language. This way, our basic assumption is that most weaknesses and resulting vulnerabilities are similar to those in other languages, potentially appearing in an Ethereum-specific form. Thus, the most promising way to create a quality assurance process for Solidity is adapting existing technologies to the peculiarities of Ethereum and, in particular, Solidity.

This Thesis proposed and discussed our approach towards assessing and improving smart contract security by dealing only with Solidity Versions 0.5 and upper. Our approach has its foundation in three main key points: *overcoming the language evolution, assessing smart contract security, and improving smart contract security.*

At first, this Thesis focused on the identification of vulnerability-related weaknesses to Solidity smart contracts. Then, we presented a general-purpose classification of 32 Solidity-specific vulnerabilities, based on CWE general software weaknesses categories (10), to overcome the language evolution dependence. This way, we provided a Solidity fault model that was the basis for the next steps in our research. Moreover, the use of CWE in abstracting a Solidity-specific vulnerability classification enriched our systematization with a widely used 'de facto' standard. In addition, this helps both software developers in limiting weaknesses explosion and their effects, and researchers in comparing Ethereum smart contracts vulnerabilities with others existing in other environments (i.e., platforms, frameworks). As the next step, we checked the similarity between Solidity (our 10 CWE-IDs) and conventional programming languages (ISO/IEC 5055:2021 – 71 CWE-IDs) in their respective fault model. Finally, to better understand the behaviour of vulnerabilities, we highlighted some relations and propagations between them.

Then, the Thesis investigated how to assess the smart contract security by applying static analysis. As a preliminary analysis, we showed that no tools of the selection cover by design the entire set of vulnerabilities, identifying vulnerability classes that escape detection by each particular SA tool. Furthermore, smart contracts showed several positives when processed by SA tools; thus, extracting a meaningful set of contracts permitted determining its ground truth.

We assessed smart contract security by computing basic statistical metrics for comparing the detection capabilities of different SA tools. Considering the anticipated vulnerability model, only in specific classes do the tools perform well (and thus are well built for those classes). This analysis serves as a guide for developers to increase the impact of tools in smart contract security. Considering tools when exposed to a generic set of smart contracts (tools as black boxes), we built a tool benchmark. Moreover, focusing on coverage, we quantitatively determined that using a single tool (even the best performing one) is not a good idea if security is the goal.

Previous findings brought directly to the analysis of the improvement of smart contract security. We investigated the coverage by using a combination of tools. This way, at the cost of increasing the false positives, we found the best n-tool combination (a combination of four tools achieves coverage of 0.9).

As the next step, we investigated several prioritization methods. Prioritization allowed us to identify which vulnerabilities that escape the tool combinations should be analyzed first due to higher severity. These analyses serve as a guide for users to make smart contracts more secure before deployment.

As a third step, we investigated where classes are most likely located into contracts. We first compared true positives and positives distribution in the pilot set and then generalized findings in the reference set. We identified where a relevant percentage of vulnerabilities is located for specific classes. Tool developers can use results to be guided to improve the tool's vulnerability detection capabilities; software developers, on the other side, can produce more secure contracts focusing on the specific areas where such vulnerabilities are most likely to be located.

Further concrete development could use the Solidity fault model as a basis for comparing Solidity-specific vulnerability categories with others affecting different smart contract languages and platforms (e.g., Hyperledger). A possible goal could be to systematize such vulnerabilities using the same categories to have a homogeneous reference that can easily be used to understand if different environments suffer from the same vulnerabilities by identifying similar behaviours occurring in such environments.

Moreover, based on static analysis results, future research can define and apply specific countermeasures against the vulnerabilities which are escaped detections. In addition, future research directions can involve the role of the contract complexity and the vulnerability and tool outcomes. Investigating which contract characteristics (e.g., software complexity) affect the tool outcomes can help software developers build more effective tools.

BIBLIOGRAPHY

[1] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system," 2008. https://bitcoin.org/bitcoin.pdf (last access: Nov. 11, 2021). Cited on pages 19, 29, 30 and 31.

[2] M. Staderini, E. Schiavone, and A. Bondavalli, "A Requirements-Driven Methodology for the Proper Selection and Configuration of Blockchains," in *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, 2018, pp. 201–206, doi: 10.1109/SRDS.2018.00031. Cited on pages 19, 29, 32 and 38.

[3] D. Siegel, "Understanding The DAO Attack," 2016. https://www.coindesk.com/learn/2016/06/25/understanding-the-dao-attack/ (last access: Nov. 08, 2021). Cited on pages 19, 21, 43, 51, 57, 64 and 74.

[4] G. Wood, "Ethereum: a secure decentralised generalised transaction ledger," 2014. Cited on pages 20, 30 and 31.

[5] Ethereum, "Solidity Documentation," *Ethereum foundation*, 2021. https://solidity.readthedocs.io/en/latest/ (last access: Mar. 01, 2021). Cited on page 20.

[6] ISO/IEC, "ISO/IEC 5055:2021 - Information technology — Software measurement — Software quality measurement — Automated source code quality measures," 2021. Cited on pages 21, 43.

[7] V. Okun, W. F. Guthrie, R. Gaucher, and P. E. Black, "Effect of static analysis tools on software security: Preliminary investigation," 2007, doi: 10.1145/1314257.1314260. Cited on page 21.

[8] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 01, no. 1, pp. 11–33, 2004. Cited on pages 24, 25, 26 and 28.

[9] A. Bondavalli, S. Bouchenak, and H. Kopetz, Eds., *Cyber-Physical Systems of Systems Foundations – A Conceptual Model and Some Derivations: The AMADEOS Legacy*, vol. 10099 LNCS. Springer Nature, 2016. Cited on page 25.

[10] W. Stallings and L. Brown, *Computer Security: Principles and Practice*, Fourth. Pearson Education, 2018. Cited on pages 28.

[11] X. Xu *et al.*, "A Taxonomy of Blockchain-Based Systems for Architecture Design," in *2017 IEEE International Conference on Software Architecture (ICSA)*, Apr. 2017, pp. 243–252, doi: 10.1109/ICSA.2017.33. Cited on pages 29, 31 and 34.

[12] M. Swan, *Blockchain for a New Economy*. O'Reilly Media, Inc., 2015. Cited on pages 29, 30 and 37.

[13] K. Wüst and A. Gervais, "Do you need a blockchain?," *2018 Crypto Val. Conf. Blockchain Technol.*, pp. 45–54, 2018, doi: 10.1109/CVCBT.2018.00011. Cited on page 31.

[14] Z. Zheng, S. Xie, H.-N. Dai, X. Chen, and H. Wang, "Blockchain Challenges and Opportunities : A Survey," *Int. J. Web Grid Serv.*, vol. 14, no. 4, pp. 352–375, 2018, doi: 10125/41338. Cited on pages 32 and 34.

[15] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends," *Proc. - 2017 IEEE 6th Int. Congr. Big Data, BigData Congr. 2017*, pp. 557–564, 2017, doi: 10.1109/BigDataCongress.2017.85. Cited on page 32.

[16] S. M. H. Bamakan, A. Motavali, and A. Babaei Bondarti, "A survey of blockchain consensus algorithms performance evaluation criteria," *Expert Syst. Appl.*, vol. 154, 2020, doi: 10.1016/j.eswa.2020.113385. Cited on page 32.

[17] C. Cachin and M. Vukolić, "Blockchain consensus protocols in the wild," *Leibniz Int. Proc. Informatics, LIPIcs*, vol. 91, 2017, doi: 10.4230/LIPIcs.DISC.2017.1. Cited on page 32.

[18] S. Bano *et al.*, "Sok: Consensus in the age of blockchains," *AFT 2019 - Proc. 1st ACM Conf. Adv. Financ. Technol.*, no. Section 4, pp. 183–198, 2019, doi: 10.1145/3318041.3355458. Cited on pages 32, 34 and 35.

[19] X. Wang, S. Duan, J. Clavin, and H. Zhang, "BFT in Blockchains: From Protocols to Use Cases," *ACM Comput. Surv.*, Nov. 2021, doi: 10.1145/3503042. Cited on page 32.

[20] ethereum.org, "Ethereum - Blocks," 2021. https://ethereum.org/en/developers/docs/blocks/ (last access: Dec. 10, 2021). Cited on page 34.

[21] ethereum.org, "The ETH2 Upgrades," 2021. https://ethereum.org/en/eth2/ (last access: Dec. 10, 2021). Cited on page 34.

[22] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," *ACM Trans. Comput. Syst.*, vol. 32, no. 4, pp. 1–45, 2015. Cited on page 34.

[23] Y. Wang *et al.*, "Formal verification of workflow policies for smart contracts in azure blockchain," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 12031 LNCS, pp. 87–106, 2020, doi: 10.1007/978-3-030-41600-3_7. Cited on pages 34, 35 and 80.

[24] D. Schwartz, N. Youngs, and A. Britto, "The Ripple protocol consensus algorithm," *Ripple Labs Inc*, pp. 1–8, 2014. Cited on page 35.

[25] M. Lokhava *et al.*, "Fast and secure global payments with stellar," *SOSP 2019 - Proc. 27th ACM Symp. Oper. Syst. Princ.*, pp. 80–96, 2019, doi: 10.1145/3341301.3359636. Cited on page 35.

[26] X. Fu, H. Wang, and P. Shi, "A survey of Blockchain consensus algorithms: mechanism, design and applications," *Sci. China Inf. Sci.*, vol. 64, no. 2, pp. 1–15, 2021, doi: 10.1007/s11432-019-2790-1. Cited on page 35.

[27] X. Li, P. Jiang, T. Chen, X. Luo, and Q. Wen, "A survey on the security of blockchain systems," *Futur. Gener. Comput. Syst.*, vol. 107, pp. 841–853, 2020, doi: 10.1016/j.future.2017.08.020. Cited on page 35.

[28] N. Szabo, "Formalizing and Securing Relationships on Public Networks," *First Monday*, vol. 2, n.9, 1997, doi: 10.5210/fm.v2i9.548. Cited on page 37.

[29] V. Buterin, "Smart contracts," *Twitter*, 2018. Cited on page 38.

[30] IBM, "What are smart contracts on blockchain?" https://www.ibm.com/dk-en/topics/smart-contracts (last access: Nov. 25, 2021). Cited on page 38.

[31] S. Wang, L. Ouyang, Y. Yuan, X. Ni, X. Han, and F. Y. Wang, "Blockchain-Enabled Smart Contracts: Architecture, Applications, and Future Trends," *IEEE Trans. Syst. Man, Cybern.*

Syst., vol. 49, no. 11, pp. 2266–2277, 2019, doi: 10.1109/TSMC.2019.2895123. Cited on pages 39 and 40.

[32] Z. Zheng *et al.*, "An overview on smart contracts: Challenges, advances and platforms," *Futur. Gener. Comput. Syst.*, vol. 105, pp. 475–491, 2020, doi: 10.1016/j.future.2019.12.019. Cited on pages 39 and 40.

[33] T. Hewa, M. Ylianttila, and M. Liyanage, "Survey on blockchain based smart contracts: Applications, opportunities and challenges," *J. Netw. Comput. Appl.*, vol. 177, no. August 2020, p. 102857, 2021, doi: 10.1016/j.jnca.2020.102857. Cited on page 40.

[34] CoinMarketCap, "Cryptocurrency market cap." https://coinmarketcap.com/ (last access: Nov. 30, 2021). Cited on page 41.

[35] ethereum.org, "Ethereum Development Documentation," 2021. https://ethereum.org/en/developers/docs/ (last access: Nov. 30, 2021). Cited on page 41.

[36] Ethereum, "Solidity Documentation," 2021. https://docs.soliditylang.org/en/latest (last access: Oct. 31, 2021). Cited on pages 41, 58 and 66.

[37] The MITRE Corporation, "CVE." https://cve.mitre.org/ (accessed Dec. 10, 2021). Cited on page 43.

[38] The MITRE Corporation, "CWE - Common Weakness Enumeration," 2021. https://cwe.mitre.org/index.html (last access: Nov. 30, 2021). Cited on pages 43, and 68.

[39] National Institute of Standards and Technology, "NATIONAL VULNERABILITY DATABASE." https://nvd.nist.gov/ (last access: Dec. 10, 2021). Cited on page 44.

[40] P. Thomson, "Static Analysis: An Introduction. The fundamental challenge of software engineering is one of complexity.," *Queue*, vol. 19, no. 4, pp. 1–13, 2021. Cited on page 45.

[41] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proc. London Math. Soc.*, pp. 230–265, 1937, doi: 10.1112/plms/s2-42.1.230. Cited on page 45.

[42] P. Emanuelsson and U. Nilsson, "A Comparative Study of Industrial Static Analysis Tools," *Electron. Notes Theor. Comput. Sci.*, vol. 217, no. C, pp. 5–21, 2008, doi: 10.1016/j.entcs.2008.06.039. Cited on pages 47 and 49.

[43] B. Chess and G. Mcgraw, "Static analysis for security," *IEEE Secur. Priv.*, vol. 2, no. 6, pp. 76–79, 2004, doi: 10.1109/MSP.2004.111. Cited on page 45.

[44] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk, "On the value of static analysis for fault detection in software," *IEEE Trans. Softw. Eng.*, vol. 32, no. 4, pp. 240–253, 2006, doi: 10.1109/TSE.2006.38. Cited on page 47.

[45] N. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (2017)*, vol. 10204 LNCS, Springer Verlag, 2017, pp. 164–186. Cited on pages 48, 57, 60, 61, 62, 63, 65 and 70.

[46] A. Dika and M. Nowostawski, "Security Vulnerabilities in Ethereum Smart Contracts," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, Jul. 2018, pp. 955–962, doi: 10.1109/Cybermatics_2018.2018.00182. Cited on pages 49 and 57.

[47] A. Mense and M. Flatscher, "Security Vulnerabilities in Ethereum Smart Contracts," in *Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services*, Nov. 2018, vol. 18, pp. 375–380, doi: 10.1145/3282373.3282419. Cited on pages 49, 57, 59 and 60.

[48] H. Hasanova, U. Baek, M. Shin, K. Cho, and M.-S. Kim, "A survey on blockchain cybersecurity vulnerabilities and possible countermeasures," *Int. J. Netw. Manag.*, vol. 29, no. 2, pp. 1–36, Mar. 2019, doi: 10.1002/nem.2060. Cited on pages 49, 60 and 62.

[49] P. Praitheeshan, L. Pan, J. Yu, J. Liu, and R. Doss, "Security analysis methods on ethereum smart contract vulnerabilities - a survey," 2020. https://arxiv.org/abs/1908.08605 (last access: Dec. 24, 2020). Cited on page 49.

[50] M. Bartoletti, S. Carta, T. Cimoli, and R. Saia, "Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact," *Futur. Gener. Comput. Syst.*, 2020, doi: 10.1016/j.future.2019.08.014. Cited on page 49.

[51] H. Chen, M. Pendleton, L. Njilla, and S. Xu, "A Survey on Ethereum Systems Security: Vulnerabilities, Attacks, and

Defenses," *ACM Comput. Surv.*, vol. 53, no. 3, pp. 1–43, Jul. 2020, doi: 10.1145/3391195. Cited on pages 49, 57, 58, 59, 60, 61, 62, 64, 65, 66, 67 and 70.

[52] R. K. McLean, "Comparing static security analysis tools using open source software," *Proc. 2012 IEEE 6th Int. Conf. Softw. Secur. Reliab. Companion, SERE-C 2012*, pp. 68–74, 2012, doi: 10.1109/SERE-C.2012.16. Cited on page 49.

[53] B. Dias, N. Ivaki, and N. Laranjeiro, "An Empirical Evaluation of the Effectiveness of Smart Contract Verification Tools," 2021. Cited on pages 52 and 53.

[54] L. Luu, D. H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the ACM Conference on Computer and Communications Security*, Oct. 2016, vol. 24-28-Octo, pp. 254–269, doi: 10.1145/2976749.2978309. Cited on pages 51 and 81.

[55] M. Di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," *Proc. - 2019 IEEE Int. Conf. Decentralized Appl. Infrastructures, DAPPCON 2019*, pp. 69–78, 2019, doi: 10.1109/DAPPCON.2019.00018. Cited on pages 51 and 79.

[56] A. Vacca, A. Di Sorbo, C. A. Visaggio, and G. Canfora, "A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges," *J. Syst. Softw.*, vol. 174, pp. 1–19, Apr. 2021, doi: 10.1016/j.jss.2020.110891. Cited on pages 51 and 79.

[57] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 Ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Jun. 2020, pp. 530–541, doi: 10.1145/3377811.3380364. Cited on pages 52, 53, 80 and 86.

[58] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, "Empirical Vulnerability Analysis of Automated Smart Contracts Security Testing on Blockchains," *Proc. 28th Annu. Int. Conf. Comput. Sci. Softw. Eng.*, pp. 103–113, 2018, doi: 10.5555/3291291.3291303. Cited on pages 52 and 53.

[59] A. Pinna, S. Ibba, G. Baralla, R. Tonelli, and M. Marchesi, "A Massive Analysis of Ethereum Smart Contracts Empirical Study and Code Metrics," *IEEE Access*, vol. 7, pp. 78194–78213, 2019, doi: 10.1109/ACCESS.2019.2921936. Cited on page 52.

[60] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Jul. 2020, pp. 415–427, doi: 10.1145/3395363.3397385. Cited on pages 52 and 53.

[61] P. Zhang, F. Xiao, and X. Luo, "A Framework and DataSet for Bugs in Ethereum Smart Contracts," *Proc. - 2020 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2020*, pp. 139–150, 2020, doi: 10.1109/ICSME46990.2020.00023. Cited on pages 52 and 53.

[62] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, B. K. Ray, and D. S. Moebus, "Orthogonal Defect Classification—A Concept for In-Process Measurements," *IEEE Trans. Softw. Eng.*, vol. 18, no. 11, pp. 943–956, 1992, doi: 10.1109/32.177364. Cited on page 52.

[63] G. Eschelbeck, "The laws of vulnerabilities: Which security vulnerabilities really matter?," *Inf. Secur. Tech. Rep.*, vol. 10, no. 4, pp. 213–219, 2005, doi: 10.1016/j.istr.2005.09.005. Cited on page 52.

[64] Q. Liu, Y. Zhang, Y. Kong, and Q. Wu, "Improving VRSS-based vulnerability prioritization using analytic hierarchy process," *J. Syst. Softw.*, vol. 85, no. 8, pp. 1699–1708, 2012, doi: 10.1016/j.jss.2012.03.057. Cited on page 53.

[65] W. Dingman *et al.*, "Defects and vulnerabilities in smart contracts, a classification using the NIST bugs framework," *Int. J. Networked Distrib. Comput.*, vol. 7, no. 3, pp. 121–132, 2019, doi: 10.2991/ijndc.k.190710.003. Cited on page 57.

[66] KingofTheEther, "King of the Ether Throne: Post-Mortem Investigation," 2016. https://www.kingoftheether.com/postmortem.html (last access: Jan. 21, 2021). Cited on pages 57, 74 and 77.

[67] Reddit, "GovernMental's 1100 ETH jackpot payout is stuck because it uses too much gas," 2016. https://www.reddit.com/r/ethereum/comments/4ghzhv/governmentals_1100_eth_jackpot_payout_is_stuck/ (last access: Jan. 21, 2021). Cited on pages 57 and 74.

[68] SmartContractSecurity, "SWC Registry," 2020. https://swcregistry.io/ (last access: Nov. 11, 2021). Cited on pages 58, 59, 62, 63, 64, 65 and 66.

[69] V. Buterin, "EIP-150: Gas cost changes for IO-heavy operations," 2016. https://eips.ethereum.org/EIPS/eip-150 (last access: Sep. 25, 2020). Cited on page 58.

[70] M. H. Swende, "EIP-1884: Repricing for trie-size-dependent opcodes," 2019. https://eips.ethereum.org/EIPS/eip-1884 (last access: Oct. 10, 2020). Cited on page 58.

[71] Ethereum, "Solidity 0.5 documentation," 2021. https://docs.soliditylang.org/en/v0.5.17/ (last access: Nov. 30, 2021). Cited on page 58.

[72] L. Breidenbach, P. Daian, A. Juels, and E. G. Sirer, "An In-Depth Look at the Parity Multisig Bug," 2017. https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/ (last access: Mar. 20, 2019). Cited on pages 60, 74, 77 and 78.

[73] StackExchange, "Griefing Attacks: Are they profitable for the attacker?," 2019. https://ethereum.stackexchange.com/questions/73261/griefing-attacks-are-they-profitable-for-the-attacker (last access: Oct. 30, 2021). Cited on page 62.

[74] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck: Static analysis of ethereum smart contracts," in *2018 ACM/IEEE 1st International Workshop on Emerging Trends in Software Engineering for Blockchain SmartCheck:*, 2018, pp. 9–16, doi: 10.1145/3194113.3194115. Cited on pages 52, 63 and 81.

[75] ConsenSys Diligence, "Ethereum Smart Contract Security Best Practices." https://consensys.github.io/smart-contract-best-practices/ (last access: Jun 20, 2021). Cited on page 65.

[76] MITRE, "CWE-1000: Research Concepts," 2021. https://cwe.mitre.org/data/definitions/1000.html (last access: Oct. 27, 2021). Cited on page 69.

[77] Ethereum, "Solidity v0.5.0 Breaking Changes," 2021. https://docs.soliditylang.org/en/latest/050-breaking-changes.html (last access: Sep. 25, 2021). Cited on page 74.

[78] Team Code4Block, "CVE-2018-17968 Exploit," 2018. https://github.com/TEAM-C4B/CVE-LIST/tree/master/CVE-2018-17987 (last access: Mar. 20, 2019). Cited on page 74.

[79] Team Code4Block, “CVE-2018-15552 Exploit,” 2021. https://github.com/TEAM-C4B/CVE-LIST/tree/master/CVE-2018-15552 (last access: Mar. 20, 2019). Cited on page 75.

[80] PeckShield, “New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018–11239),” 2018. https://peckshield.medium.com/new-burnoverflow-bug-identified-in-multiple-erc20-smart-contracts-cve-2018-11239-52cc4f821694 (last access: Nov. 18, 2021). Cited on page 76.

[81] BlockChainsSecurity, “Sell Integer Overflow,” 2018. https://github.com/BlockChainsSecurity/EtherTokens/blob/master/ETHEREUMBLACK/sell%20integer%20overflow.md (last access: Mar. 20, 2019). Cited on page 76.

[82] J. Song, “Aurora IDEX Membership(IDXM), ERC20 Token, allows attackers to acquire contract ownership (CVE-2018–10666),” 2018. https://medium.com/@jonghyk.song/aurora-idex-membership-idxm-erc20-token-allows-attackers-to-acquire-contract-ownership-1ff426cee7c6 (last access: Mar. 20, 2020). Cited on page 77.

[83] X. Liang, “Support new solc & UTF-8,” *2020.* https://github.com/enzymefinance/oyente/pull/406 (last access: Nov. 08, 2021). Cited on page 80.

[84] R. Norvill, B. B. F. Pontiveros, R. State, and A. Cullen, “Visual emulation for Ethereum’s virtual machine,” *IEEE/IFIP Netw. Oper. Manag. Symp. Cogn. Manag. a Cyber World, NOMS 2018*, pp. 1–4, 2018, doi: 10.1109/NOMS.2018.8406332. Cited on page 80.

[85] Y. Zhou, D. Kumar, S. Bakshi, J. Mason, A. Miller, and M. Bailey, “Erays: Reverse engineering Ethereum’s opaque smart contracts,” in *Proceedings of the 27th USENIX Security Symposium*, 2018, pp. 1371–1385. Cited on page 80.

[86] J. Frank, C. Aschermann, and T. Holz, “ETHBMC: A bounded model checker for smart contracts,” *Proc. 29th USENIX Secur. Symp.*, pp. 2757–2774, 2020. Cited on page 80.

[87] I. Grishchenko, M. Maffei, and C. Schneidewind, “EtherTrust: Sound Static Analysis of Ethereum bytecode,” *Tech. Univ. Wien, Tech. Rep*, pp. 1–41, 2018. https://www.netidee.at/sites/default/files/2018-07/staticanalysis.pdf (last access: 25 Mar, 2020). Cited on page 80.

[88] E. Albert, P. Gordillo, B. Livshits, A. Rubio, and I. Sergey, "EthIR: A Framework for High-Level Analysis of Ethereum Bytecode," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 11138 LNCS, pp. 513–520, 2018, doi: 10.1007/978-3-030-01090-4_30. Cited on page 80.

[89] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "EThor: Practical and Provably Sound Static Analysis of Ethereum Smart Contracts," 2020, doi: 10.1145/3372297.3417250. Cited on page 80.

[90] T. Chen *et al.*, "GasChecker: Scalable Analysis for Discovering Gas-Inefficient Smart Contracts," *IEEE Trans. Emerg. Top. Comput.*, pp. 1–1, 2020, doi: 10.1109/TETC.2020.2979019. Cited on page 80.

[91] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *SANER 2017 - 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2017, pp. 442–446, doi: 10.1109/SANER.2017.7884650. Cited on page 80.

[92] E. Hildenbrandt *et al.*, "KEVM: A complete formal semantics of the ethereum virtual machine," *Proc. - IEEE Comput. Secur. Found. Symp.*, vol. 2018-July, pp. 204–217, 2018, doi: 10.1109/CSF.2018.00022. Cited on page 80.

[93] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: surviving out-of-gas conditions in Ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–27, 2018, doi: 10.1145/3276486. Cited on page 80.

[94] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," *ACM Int. Conf. Proceeding Ser.*, pp. 653–663, 2018, doi: 10.1145/3274694.3274743. Cited on page 80.

[95] M. Mossberg *et al.*, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," *Proc. - 2019 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2019*, pp. 1186–1189, 2019, doi: 10.1109/ASE.2019.00133. Cited on page 80.

[96] Patrick Ventuzelo, "Octopus - repository." https://github.com/pventuzelo/octopus (last access: Nov. 10, 2021). Cited on page 80.

[97] M. Suiche, "Porosity: A decompiler for blockchain-based smart contracts bytecode," 2017. Cited on page 80.

[98] Crytic, "Rattle - repository." https://github.com/trailofbits/rattle (last access: Nov. 10, 2021). Cited on page 80.

[99] E. Zhou *et al.*, "Security Assurance for Smart Contract," *2018 9th IFIP Int. Conf. New Technol. Mobil. Secur. NTMS 2018 - Proc.*, vol. 2018-Janua, pp. 1–5, 2018, doi: 10.1109/NTMS.2018.8328743. Cited on page 80.

[100] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "sCompile: Critical Path Identification and Analysis for Smart Contracts," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 11852 LNCS, 2019, pp. 286–304. Cited on page 80.

[101] C. Peng, S. Akca, and A. Rajan, "SIF: A Framework for Solidity Contract Instrumentation and Analysis," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2019, pp. 466–473, doi: 10.1109/APSEC48747.2019.00069. Cited on page 80.

[102] Z. Gao, V. Jayasundara, L. Jiang, X. Xia, D. Lo, and J. Grundy, "SmartEmbed: A Tool for Clone and Bug Detection in Smart Contracts through Structural Code Embedding," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2019, pp. 394–397, doi: 10.1109/ICSME.2019.00067. Cited on page 80.

[103] S. Bragagnolo, H. Rocha, M. Denker, and S. Ducasse, "SmartInspect: solidity smart contract inspector," in *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, Mar. 2018, vol. 2018-Janua, pp. 9–18, doi: 10.1109/IWBOSE.2018.8327566. Cited on page 80.

[104] S. Akca, A. Rajan, and C. Peng, "SolAnalyser: A Framework for Analysing and Testing Smart Contracts," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2019, pp. 482–489, doi: 10.1109/APSEC48747.2019.00071. Cited on pages 52 and 80.

[105] R. Revere, "Solgraph - repository." https://github.com/raineorshine/solgraph (last access: Nov. 10, 2021). Cited on page 80.

[106] Protofire, "Solhint - repository." https://github.com/protofire/solhint (last access: Nov. 10, 2021). Cited on page 80.

[107] P. Hegedűs, "Towards Analyzing the Complexity Landscape of Solidity Based Ethereum Smart Contracts," *Technologies*, vol. 7,

no. 1, p. 6, 2019, doi: 10.3390/technologies7010006. Cited on page 80.

[108] Á. Hajdu and D. Jovanović, "solc-verify: A Modular Verifier for Solidity Smart Contracts," in *Springer*, vol. 12031 LNCS, Springer, 2020, pp. 161–179. Cited on page 80.

[109] L. Brent *et al.*, "Vandal: A scalable security analysis framework for smart contracts," 2018. https://arxiv.org/abs/1809.03981v1 (last access Jan. 20, 2021). Cited on page 80.

[110] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing Safety of Smart Contracts.," 2018. Cited on page 80.

[111] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2018, pp. 67–82, doi: 10.1145/3243734.3243780. Cited on page 80.

[112] SRILAB EthZurich, "Securify - repository." https://github.com/eth-sri/securify (last access: Nov. 10, 2021). Cited on page 81.

[113] SRI Lab. EthZurich, "Securify v2.0 - repository." https://github.com/eth-sri/securify2 (last access: Nov. 10, 2020). Cited on page 81.

[114] J. Feist, G. Greico, and A. Groce, "Slither: A static Analysis Framework for Smart Contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in SOftware Egineering for Blockchain*, Mar. 2019, pp. 8–15, doi: 10.1109/SANER.2017.7884650. Cited on page 81.

[115] Crytic, "Slither 0.7.1 - repository." https://github.com/crytic/slither/tree/0.7.1 (last access: Nov. 10, 2021). Cited on page 81.

[116] SmartDec, "SmartCheck - repository." https://github.com/smartdec/smartcheck (last access: Nov. 10, 2021). Cited on page 81.

[117] Ethereum, "Remix Project - repository." https://github.com/ethereum/remix-project (last access: Nov. 10, 2021). Cited on page 81.

[118] ConsenSys Diligence, "Mythril 0.22.17 - repository." https://github.com/ConsenSys/mythril/tree/v0.22.17 (last access: Nov. 10, 2021). Cited on page 81.

[119] 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*, Dec. 2018, pp. 664–676, doi: 10.1145/3274694.3274737. Cited on page 81.

[120] Enzyme Finance, "Oyente - repository." https://github.com/enzymefinance/oyente (last access: Nov. 10, 2021). Cited on page 81.

[121] C. F. Torres, "Osiris - repository." https://github.com/christoftorres/Osiris (last access Nov. 10, 2021). Cited on page 81.

[122] C. F. Torres, M. Steichen, and R. State, "The art of the scam: Demystifying honeypots in ethereum smart contracts," 2019. https://www.usenix.org/conference/usenixsecurity19/presentation/ferreira (last access: Dec. 24, 2020). Cited on page 81.

[123] C. Torres, "HoneyBadger - repository." https://github.com/christoftorres/HoneyBadger (last access: Nov. 10, 2021). Cited on page 81.

[124] Staderini Mirko, "Archive of support files," 2022. https://github.com/mstad/Miscellanea/blob/main/Staderini_Thesis_Archive.zip. Cited on pages 23, 71, 82, 85, and 108.

[125] F. Vogelsteller and V. Buterin, "EIP-20: ERC-20 Token Standard," 2015. https://eips.ethereum.org/EIPS/eip-20 (last access: Aug. 10, 2020). Cited on page 84.

[126] Ethereum, "Etherscan - The Ethereum Blockchain Explorer." https://etherscan.io/ (last access: Nov. 08, 2021). Cited on page 85.

[127] G. A. Oliva, A. E. Hassan, and Z. M. Jiang, "An exploratory study of smart contracts in the Ethereum blockchain platform," *Empir. Softw. Eng.*, vol. 25, no. 3, pp. 1864–1904, May 2020, doi: 10.1007/s10664-019-09796-5. Cited on page 85.

[128] S. V. Stehman, "Selecting and interpreting measures of thematic classification accuracy," *Remote Sens. Environ.*, vol. 62, no. 1, pp. 77–89, 1997, doi: 10.1016/S0034-4257(97)00083-7. Cited on page 86.

[129] K. H. Brodersen, C. S. Ong, K. E. Stephan, and J. M. Buhmann, "The balanced accuracy and its posterior distribution," *Proc. - Int. Conf. Pattern Recognit.*, pp. 3121–3124, 2010, doi: 10.1109/ICPR.2010.764. Cited on page 88.

[130] S. Chulani and B. Boehm, "Modeling Software Defect Introduction and Removal: COQUALMO (COnstructive QUALity MOdel)," *USC-CSE Tech. Rep.*, pp. 99–510, 1999. Cited on page 97.

[131] F. Di Giandomenico and L. Strigini, "Adjudicators for diverse-redundant components," in *Proceedings Ninth Symposium on Reliable Distributed Systems*, 1990, pp. 114–123, doi: 10.1109/RELDIS.1990.93957. Cited on page 100.

[132] FIRST.Org, "Common Vulnerability Scoring System SIG," 2021. https://www.first.org/cvss/ (accessed Nov. 11, 2021). Cited on page 101.

[133] MITRE Corporation, "Common Attack Pattern Enumeration and Classification," 2021. https://capec.mitre.org/ (last access: Nov. 11, 2021). Cited on page 101.

[134] MITRE Corporation, "Common Weakness Scoring System," 2018. https://cwe.mitre.org/cwss/cwss_v1.0.1.html (last access: Nov. 11, 2021). Cited on pages 101, 103 and 103.

[135] NIST, "NVD Vulnerability Severity Ratings," 2020. https://nvd.nist.gov/vuln-metrics/cvss (last access: Nov. 11, 2021). Cited on page 103.

[136] P. Nunes, I. Medeiros, J. C. Fonseca, N. Neves, M. Correia, and M. Vieira, "Benchmarking Static Analysis Tools for Web Security," *IEEE Trans. Reliab.*, vol. 67, no. 3, pp. 1159–1175, 2018, doi: 10.1109/TR.2018.2839339. Cited on page 51.

[137] MITRE, "CWE Mapping & Navigation Guidance," 2018. https://cwe.mitre.org/documents/cwe_usage/mapping_navigation.html (last access: Nov. 11, 2021). Cited on page **Error! Bookmark not defined.**.

[138] Center for Assured Software - NSA, "Juliet Test Suite v 1.2 for Java - User Guide," 2012. Cited on page 55.