Indonesian Journal of Electrical Engineering and Computer Science
Vol. 38, No. 1, April 2025, pp. 657~667
ISSN: 2502-4752, DOI: 10.11591/ijeecs.v38.i1.pp657-667

SmartSentry: a comprehensive framework for automated vulnerability discovery in Ethereum smart contracts

Oualid Zaazaa, Hanan El Bakkali
Rabat IT Center, Smart Systems Laboratory, ENSIAS, Mohammed V University, Rabat, Morocco

Article Info

Article history:

Received Mar 26, 2024
Revised Oct 7, 2024
Accepted Oct 30, 2024

Keywords:

Abstract syntax trees analysis
Blockchain security
Dataflow analysis
Smart contract vulnerability
Static analysis

ABSTRACT

In the realm of decentralized applications, smart contracts play a pivotal role in managing an extensive array of digital assets within blockchain networks. Ensuring the security of these digital assets hinges upon the adept detection of vulnerabilities present within smart contracts. Extensive research efforts have scrutinized and elucidated numerous smart contract vulnerabilities. However, certain vulnerabilities, including signature malleability, hash collision, and inconsequential code segments, remain relatively unexplored and devoid of dedicated detection tools. In response to this research gap, this paper addresses these three previously understudied vulnerabilities. We contribute to the field by creating a labeled dataset comprising vulnerable smart contracts. This dataset serves as a valuable resource for further scientific inquiries, enabling the testing and validation of various detection frameworks. Additionally, we present SmartSentry a static vulnerability detection framework capable of identifying these vulnerabilities. Using both dataflow and control flow analysis, our framework exhibits exceptional performance, successfully identifying labeled vulnerabilities and real-world vulnerabilities within production smart contracts with speed and efficiency. These efforts collectively enhance our understanding of smart contract vulnerabilities and contribute to the broader advancement of blockchain security.

Corresponding Author:

Oualid Zaazaa
Rabat IT Center, Smart Systems Laboratory, ENSIAS, Mohammed V University
Rabat-10112, Morocco
Email: oualid_zaazaa@um5.ac.ma

1. INTRODUCTION

Smart contracts are self-executing contracts with the terms of the agreement directly written into code, operating on blockchain technology. When predefined conditions are met, the code executes corresponding actions, such as transferring assets or issuing penalties. This automation enhances transparency, efficiency, and security by eliminating human error and reducing the need for trust between parties [1]. While smart contracts offer remarkable advantages, they are not without their share of challenges [2]. One of the most pressing concerns relates to security vulnerabilities. As with any piece of software, smart contracts can contain flaws that malicious actors may exploit to gain unauthorized access or manipulate the contract's intended behavior [3]. Notably, the 2016 attack [4] on "The DAO," a decentralized autonomous organization built on the Ethereum platform, stands as a prominent example of the potential risks associated with smart contracts. Attackers managed to exploit vulnerabilities in The DAO's smart contract code, resulting in the misappropriation of more than 3.6 million ethers from the organization. Another instance highlighting the vulnerability of smart contracts transpired in 2018 with the "BEC Token" incident [5].

In this case, an attacker leveraged an integer overflow vulnerability in the BEC smart contract to generate tokens indefinitely, ultimately leading to a substantial loss of approximately 6 billion tokens.

Due to both their immutability feature and the managed sensitive data, smart contract vulnerabilities should be checked and fixed before a production deployment. The rise in their complexity make manual vulnerability detection of smart contract not efficient and require to be completed by an automated scan [6] Therefore, multiple previous researchs [7]–[11] have been performed to build frameworks that automatically discover those vulnerabilities in the development phase. Feist *et al.* [12] have built slither, a static smart contract analyser that is capable of detecting vulnerabilities like shadowing [3], uninitialized variables [3], reentrancy [13] and a variety of other known security issues, such as suicidal contracts, locked ether, or arbitrary sending of ether. Grech *et al.* [14] built MadMax, a static analysis framework capable of detecting out-of-gaz related vulnerabilities [3]. Nguyen *et al.* [15] in 2020 have built sFuzz a dynamic analysis framework capable of detecting vulnerabilities like reentrancy, timestamp dependency [3], block number dependency [3], integer overflow [3]. Ren *et al.* [16] have also built a static vulnerability detection framework called Solidifier capable of detecting vulnerabilities like, reentrancy, timestamp dependency, front running, integer overflow, and others.

However, to our best knowledge and according to a systematic literature review [17] we have conducted before, none of those reaserch have ever studied signature malleability (SWE-117) [3] vulnerability that allow forging a valid signature starting from a valid one, hash collision with multiple variables of varying lengths (SWE-133) [3] that allow attacker to generate similar hash with different data, and code with no effect (SWE-135) [3] where a segment of code within the smart contract is implemented without achieving the intended functionality.

Therefore, the overarching goal of this paper is to characterize, and mitigate the SWE-117, SWE-133 and the SWE-135 vulnerabilities that could be made by developers. As a concrete instance of this problem, we focus on EVM based blockchains with experimentation beeing made on Ethereum deployed smart contracts. To date, Ethereum blockchain hosts more than 61 million [18] deployed smart contract and has more 200k smart contract deployed each day [19]. Thus, to mitigate those vulnerabilities in early stage of smart contract development process we introduces SmartSentry a novel framework designed for the systematic detection of vulnerabilities within smart contracts. SmartSentry is constructed upon an analytical foundation that incorporates abstract syntax trees (AST), control flow analysis (CFA), and data flow analysis (DFA) techniques. It is structured into three distinct components, each serving a specific function. The first component is dedicated to the compilation phase, responsible for the extraction of pertinent features from the AST and control flow graph (CFG). The second component undertakes a comprehensive analysis of these features, employing a predefined set of patterns conforming to AST and CFG structures. In instances where the previous stages fail to render a decisive determination regarding the presence of vulnerabilities, SmartSentry seamlessly directs the extracted features to a data flow analyzer, facilitating the application of supplementary analytical rules, thereby minimizing the occurrence of false positives.

To successfuly reach our goal, this paper is divided into 4 sections. Section 2, will try to dissect the three previously mentioned vulnerabilities to understand the different technical details that will allow us to correctly identify the root cause behind each one of them. Section 3, gives a detailed overview about some related works while categorizing them into static and dynamic approaches. Sections 4, identify and explain the root cause of the three vulnerabilities while introducing SmartSentry architecture and internals. Section 5 explain the different steps of the performed experimentation and discuss the final results and accuracy of SmartSentry. In summary, this paper makes the following key contributions: i) dissecting the root cause of three none studied smart contract vulnerabilities; ii) build a dataset of smart contracts vulnerable to SWE-117, SWE-133 and the SWE-135; and iii) introduce SmartSentry a new modular smart contract vulnerability scanner capable of detecting SWE-117, SWE-133 and the SWE-135.

2. SMART CONTRACT VULNERABILITIES

Within this section, we delve into a comprehensive analysis of three distinct categories of vulnerabilities, which we subsequently incorporate into our established framework. The rationale guiding the selection of these particular vulnerabilities is firmly grounded in the findings emanating from our antecedent systematic literature review (SLR) [17]. Our systematic literature review (SLR) findings clearly indicate that these three vulnerabilities have not been subject to thorough academic scrutiny until now. Furthermore, the comprehensive survey has revealed a notable absence within the corpus of scientific literature, as no existing framework have, until the day of writing this paper, been introduced to effectively discern or address these specific vulnerabilities. The three primary smart contract vulnerabilities, which constitute the focus of our research, are as follows.

2.1. Signature malleability (SWE-117)

Ethereum, a prominent blockchain platform, relies on the elliptic curve digital signature algorithm (ECDSA) [1] to ensure the authenticity and integrity of digital signatures. While ECDSA is a robust cryptographic mechanism, a significant vulnerability known as "Signature malleability" has been identified in the context of Ethereum smart contracts. This vulnerability arises when smart contracts do not employ ECDSA properly and, consequently, validate signatures inadequately. Signature malleability allows malicious actors to subtly modify existing signatures without rendering them invalid, effectively bypassing signature validity checks and compromising the integrity of the smart contract.

It is crucial to note that attackers do not gain access to the signer's private key during this process. Instead, they manipulate the properties of an already used signature to generate an alternative but equally valid signature. This manipulation facilitates their ability to circumvent security measures implemented within the smart contract.

To comprehend the source of this vulnerability, it is essential to delve deeper into the workings of ECDSA, particularly within the Ethereum context. ECDSA signatures rely on the SECP256k1 elliptic curve, characterized by the equation $y^2=x^3+7$. Notably, this curve exhibits a symmetrical nature over the x-axis, a key factor contributing to signature malleability.

In ECDSA, signatures are typically represented as values (r, s), often accompanied by an additional value denoted as "v," referred to as the recovery value. The recovery value plays a pivotal role in determining the public key from the value "r." The public key corresponds to another point on the elliptic curve. In the absence of the recovery value "v," two candidate public keys are generated, each symmetrically reflecting the other over the x-axis. To illustrate, given an ECDSA signature (r, s, v) and a curve point "P" corresponding to one of the candidate public keys, the other candidate public key is derived as the point (x_P, y_P'), where "x_P" represents the x-coordinate of "P," and "y_P'" is the negation of the y-coordinate of "P." The signature components (r, s) are defined as follows: "r" represents the x-coordinate of a point generated during the signing process. It is generated through a complex algorithm involving a random value "k" and the elliptic curve's generator point "G." Typically, "k" is deterministically calculated using the private key and the message to be signed. "s" is computed using the formula $s=k^{-1}*(e+d*r)$ mod "n," where "e" is the hash of the message to be signed, and "d" denotes the signer's private key.

The vulnerability [3] arises from the x-axis symmetry inherent in the elliptic curve. If (r, s) constitutes a valid signature, then (r, -s mod "n") is also a valid signature. This malleability of ECDSA signatures underscores the need for careful implementation and validation of signatures within Ethereum smart contracts to mitigate the risk of unintended alterations and potential security breaches.

2.2. Hash collision with multiple variables of varying lengths (SWE-133)

The second vulnerability we will focus on in this research is the SWE-133 [3]. when utilizing the abi.encodePacked() function with two or more string or array parameters positioned side by side, presents a notable concern. Importantly, this susceptibility arises primarily when these strings or arrays are contiguous, regardless of the presence of other parameters at the extremities. The abi.encodePacked() function, a fundamental feature within the Solidity programming language, is responsible for concatenating input parameters according to a predefined sequence, treating them as discrete components. It is imperative to note that this issue is not novel and is not confined solely to the Solidity framework [20]. The vulnerability emanates from the intrinsic characteristics of hash functions and the way data are concatenated to each other.

The ramifications of hash collisions within the context of smart contracts are multifaceted. Notably, they can undermine access control and authentication mechanisms by enabling disparate inputs to yield identical hash outputs, inadvertently granting unauthorized access. In data structures like hash tables and mappings, hash collisions have the potential to disrupt data retrieval and storage operations, leading to issues related to data integrity, unauthorized data manipulation, or erroneous system behavior.

2.3. Code with no effect (SWE-135)

The final vulnerability, herein referred to as "code with no effect" and codified as SWE-135 [3], merits in-depth analysis. This vulnerability pertains to the composition of code segments within smart contracts that neither induce any substantive modification to the program's state nor exhibit any influence on the execution path. In practical terms, the presence of such code segments may mislead the coder into perceiving that a specific operation or condition is executed when, in reality, it remains dormant or with no effect. This vulnerability raises concerns not only within the Solidity framework but also across a spectrum of other software technologies.

A code with no effect, as encountered within solidity or analogous programming languages, may manifest as a redundant conditional statement that is always true or always false, thereby having no real bearing on the program's logic. An illustrative example may include a control structure that ostensibly

evaluates a certain condition but is devoid of any substantive functionality. Such a code segment can, in some instances, be a vestige of debugging or experimentation, inadvertently left behind in the final codebase.

The potential impact of this vulnerability extends to several facets. In financial smart contracts, for instance, where the code with no effect may falsely convey a transaction's execution without any substantive changes, it can lead to erroneous financial operations. This could inadvertently result in misallocation of funds, negation of contractual obligations, or even facilitate unauthorized actions. Furthermore, in systems where computational resources are a premium, these extraneous code segments may unduly consume resources, resulting in inefficiencies and increased operational costs. Thus, recognizing and addressing the issue of "code with no effect" is imperative not only for the integrity and efficiency of smart contracts but also for safeguarding against potential security and financial risks, which transcend the confines of Solidity to impact a broader technological landscape.

3. RELATED WORK

This section will explore influential research endeavors that have harnessed dynamic and static analysis techniques to build frameworks designed to discover different type of smart contract vulnerabilities.

3.1. Dynamic vulnerability detection technique

Dynamic analysis can be classified into two principal categories: black box analysis and gray box analysis. In black box analysis, the analyzer operates without prior knowledge of the target source code or underlying architecture. This approach entails subjecting the program to a barrage of diverse inputs with the explicit objective of causing program crashes or failures or behave in unpredictable way. Conversely, gray box (hybrid) analysis combines dynamic analysis with static analysis, either automatically or through manual intervention, with the aim of enhancing analysis efficacy and reducing overall analysis time [21].

Numerous researchers have embraced the methodology of vulnerability detection, employing specialized techniques such as Fuzzing, data flow analysis, and others. Notably, Christof *et al.* [22] have incorporated dynamic analysis within their framework, ÆGIS. Within this framework, the dynamic control flow analysis technique is harnessed to construct a call tree based on instructions processed by the tool's interpreter component. Subsequently, the dynamic taint analysis technique is applied to trace the movement of data between instructions. Nguyen *et al.* [15] as detailed in the sFuzz paper, have also employed this technique as a cornerstone in the development of their sFuzz framework. This tool strategically leverages a dynamic analysis technique known as feedback-guided adaptive fuzzing. A similar approach was used also by Liu *et al.* [23] to build ReGuard. ReGuard constitutes a fuzzing-based analyzer that has been developed with the specific purpose of automating the detection of reentrancy vulnerabilities within Ethereum smart contracts.

The utilization of dynamic analysis techniques in the evaluation of smart contracts presents several inherent limitations. Firstly, the dynamic analysis may not provide exhaustive coverage of all possible program paths and inputs, thus leaving unexplored vulnerabilities undetected. Additionally, the technique is susceptible to the generation of false positives and false negatives, whereby it may erroneously identify non-existent vulnerabilities or overlook actual security flaws. Furthermore, dynamic analysis can be resource-intensive, consuming considerable computational power and time, particularly when assessing complex and extensive smart contracts [21].

3.2. Static vulnerability detection technique

Static analysis, a pivotal process in the domain of system evaluation, is rooted in the comprehensive examination of a system's structure, content, and documentation, all without the need for program execution. This approach is grounded in the systematic collection of program source code components, including source files, libraries, and dependencies, for in-depth scrutiny and analysis. The overarching goal of static analysis is to unveil latent defects and irregularities that possess the potential to culminate in vulnerabilities, thereby fortifying the robustness and security of software systems. Traditionally, static analysis thrives in contexts where access to the source code is readily available, allowing for a detailed assessment of the program's design and logic. It is a proactive method, often employed during the software development phase, enabling early identification and rectification of issues. Furthermore, static analysis transcends the confines of source code assessment and can be judiciously employed to scrutinize binary applications through the meticulous examination of their assembly language [24]. This versatility underscores the adaptability of static analysis as a comprehensive software quality assurance tool. As static analysis techniques continue to evolve, a myriad of approaches and methodologies are being explored and developed to augment their effectiveness. The landscape of static analysis encompasses a diverse range of tools and practices, including abstract interpretation, data flow analysis, and formal methods, all of which are designed to bolster the identification and mitigation of vulnerabilities within software systems. Notably, Feist *et al.* [12] have built a framework

called Slither. Slither conducts a comprehensive static analysis of contracts through a multistage process. It initiates the analysis by taking the solidity AST, initially generated by the Solidity compiler from the source code of the contract, as input. In the first stage of analysis, Slither extracts critical information, including the contract's inheritance hierarchy, the control flow graph (CFG), and the list of expressions. Subsequently, the entire contract code is transformed into SlithIR, an internal representation language unique to Slither. SlithIR utilizes static single assignment (SSA) principles, streamlining the execution of diverse code analyses. Tikhomirov *et al.* [25] have also worked on a framework capable of detecting smart contract vulnerabilities called SmartCheck. SmartCheck is a Java-based static analysis tool specifically designed for Ethereum smart contracts. It is analysis process entails both lexical and syntactical examinations of Solidity source code. To accomplish this, SmartCheck employs ANTLR in combination with a customized Solidity grammar, facilitating the generation of an XML parse tree that serves as an intermediate representation (IR). Vulnerability patterns are identified through the utilization of XPath queries on this IR. As a result, SmartCheck delivers comprehensive coverage, ensuring that the entire code under analysis is entirely translated into the IR, enabling accessibility to all its constituent elements through XPath matching. Other researchers like Loi *et al.* [26] have built their framework (Oyente) around another static analysis technique called symbolic execution. A symbolic execution employs a representation wherein program variable values are expressed as symbolic expressions of the input's symbolic values. Each symbolic path is accompanied by a corresponding path condition, which is essentially a formula constructed from accumulated constraints that must be met by the symbolic inputs for the execution to traverse that path. In cases where a path's condition becomes unsatisfiable, the path is deemed infeasible. Conversely, when the path condition remains satisfiable, the path itself is considered feasible.

4. METHOD

Given the immutability of smart contracts and limitations of dynamic analysis, using this technique post-deployment is not a viable option for vulnerability detection. Consequently, we opted for static analysis within our framework to identify vulnerabilities during the early stages of the development lifecycle, prior to deployment. However, to develop robust algorithms capable of detecting the three targeted smart contract vulnerabilities, a comprehensive understanding of these vulnerabilities is essential. In this section, we elucidate the root causes and specific details of each vulnerability detector.

4.1. Vulnerabilities root cause

In order to implement an automated detection mechanism for the vulnerabilities under consideration, it is imperative to comprehensively comprehend the root causes of the identified issues. Primarily, we address the matter of signature malleability (SWE-117). The root cause of the signature malleability issue lies in the system's approach to handling elliptic curve digital signature algorithm (ECDSA) signatures. In the ECDSA, a signature comprises two components: 'r' and 's'. For a signature to be considered non-malleable, it is essential that these components are uniquely defined and adhere to specific constraints. The specific vulnerability arises due to the system's failure to enforce a critical condition on the 's' component of the signature. According to the standards, for a signature to be unique and hence non-malleable, 's' should be in the lower half of the curve's order, which means it should be higher than the value 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7 357A4501DDFE92F46681B20A0. This threshold is half the order of the secp256k1 curve used in Ethereum's ECDSA. When the system does not enforce this condition, it becomes possible to generate multiple valid signatures for the same transaction. To detect this vulnerability we have built an algorithm [27] based on the analysis of AST and dataflow.

The second vulnerability, identified as "Hash collision with multiple variables of varying lengths," originates from a fundamental cause associated with the simultaneous utilization of a minimum of two strings or arrays during the invocation of the "abi.encodePacked()" function. The said function orchestrates a direct concatenation of disparate parameters devoid of any padding. This absence of padding introduces the possibility of generating identical byte sequences with entirely disparate inputs. In essence, the vulnerability arises due to the intrinsic nature of the "abi.encodePacked()" function, wherein its concatenation mechanism lacks a uniform padding strategy, thereby facilitating the generation of colliding hashes from distinct input configurations. A judicious examination of this vulnerability underscores the imperative of adopting enhanced strategies for parameter encoding and concatenation to mitigate the risks inherent in hash collisions arising from variable-length input combinations. To discover this vulnerability another algorithm was built based on AST only as the detection doesn't require any data flow analysis [28].

The "Code with no effect" vulnerability poses a considerable challenge for automated detection due to its inherent lack of a specific pattern associated with its root cause in most instances. The manifestation of this vulnerability can be as elementary as a condition that perpetually evaluates to either true or false, or the absence of a crucial parameter requisite for the execution of an intended action within the application, among

other possibilities. To address this complexity, our research focuses on three primary instances of this vulnerability, each exhibiting discernible root cause patterns. These instances include a condition that invariably evaluates to true or false, the utilization of ".call" without an accompanying empty string parameter, and the occurrence of a condition written outside a block controller (such as "if" or "for" …). The scrutiny of these specific patterns aims to enhance the precision and efficacy of automated detection mechanisms for the "Code with No Effect" vulnerability. However, as code with no effect vulnerability could rize from any kind of source code, we have focused mainly on three instances of this vulnerability (event call without "()" and without "emit", Binary Operation outside block, call.value()() in pragma version less than 6) [29].

4.2. Framework design

The architecture of SmartSentry is characterized by three interdependent components as shown in Figure 1: the pre-compiler, compiler, and analyzer, each contributing to a holistic and thorough smart contract analysis pipeline. The pre-compiler module, positioned as the inaugural stage, serves as the entry point for the analysis process. It accommodates diverse inputs, allowing users to specify either a GitHub URL or a smart contract address on the Ethereum blockchain. The flexibility in input sources caters to various development and deployment scenarios. Upon receiving the input, the pre-compiler diligently undertakes the task of retrieving the specified codebase. This not only includes the primary smart contract but also extends to encompass all associated libraries, packages, and relevant code dependencies. The comprehensive inclusion of these elements ensures a self-contained and cohesive environment for subsequent compilation stages. The goal is to preemptively address any potential compilation challenges arising from dependencies, thereby enhancing the efficiency of the overall analysis process. Moving forward, the compiler, as the core processing unit, takes charge of the compilation process. Beyond the conventional compilation task, it plays a pivotal role in generating crucial artifacts that form the basis of subsequent analyses. Specifically, the compiler extracts both the AST and the CFG from the compiled code. These representations provide a high-level abstraction of the smart contract's structure and the sequence of control flow, offering valuable insights into the code's intricacies.

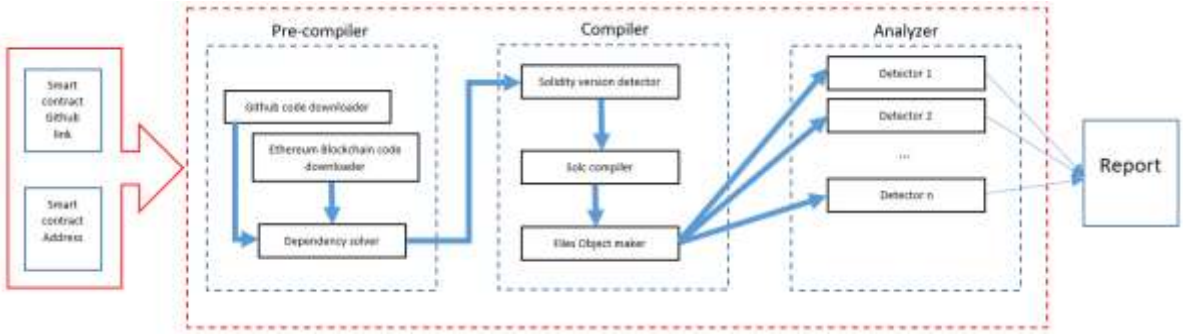

Figure 1. Framework overview

The extracted AST and CFG, now enriched with semantic information, are then transmitted to the static analyzer. This component constitutes the heart of SmartSentry, employing static analysis techniques such as AST and control flow analysis and dataflow analysis. The static analyzer meticulously examines the received artifacts, conducting an in-depth analysis to detect potential security vulnerabilities. It scrutinizes the code for patterns indicative of common vulnerabilities, potential exploits, and deviations from best coding practices. The analyzer has been systematically designed with a focus on modularity, adhering to a structured approach that isolates each vulnerability's detection mechanism from others within SmartSentry. This deliberate separation of detectors ensures a modular and flexible architecture, facilitating seamless integration and scalability for future enhancements. The segmentation of vulnerability detectors allows for targeted improvements or the addition of new detectors without necessitating extensive modifications to the existing framework. This design rationale is founded on the principle of providing an adaptable framework where individuals can effortlessly construct detectors tailored to specific vulnerabilities. Leveraging the built-in functionalities for AST analysis or dataflow analysis, stakeholders have the capability to construct specialized detectors in accordance with the unique requirements of distinct vulnerabilities. This modular design philosophy not only enhances the extensibility and versatility of the framework but also fosters a collaborative environment conducive to continual advancements in smart contract security analysis.

4.3. Experimental design

In this section, we outline our experimental methodology, which was designed to achieve specific objectives central to the evaluation of SmartSentry. Our primary objective was to test the effectiveness of SmartSentry in identifying known vulnerabilities within smart contracts and simultaneously validate its conceptual foundations. To accomplish this, we deliberately crafted vulnerable smart contracts, each exemplifying a known vulnerability. Additionally, we assembled a diverse dataset of 1,000 smart contracts [30], encompassing a wide range of code scenarios. This dataset served as the basis for assessing the framework's performance in a real-world context. The methodology was structured into three distinct phases, each serving a unique purpose: Phase 1-dataset preparation: this initial phase focused on dataset preparation. We curated a dataset comprising intentionally crafted vulnerable smart contracts [31] and an additional set of 1,000 diverse smart contracts [30]. The former was instrumental in controlled testing, while the latter introduced real-world variability into our evaluation; Phase 2-vulnerable smart contract testing: the second phase involved the systematic execution of SmartSentry on the dataset of intentionally crafted vulnerable smart contracts. This controlled environment allowed us to assess the framework's ability to detect and mitigate known vulnerabilities effectively; and Phase 3-performance testing: the third and final phase centered on evaluating the real-world performance of SmartSentry. We deployed the framework on the dataset comprising 1,000 smart contracts, drawn from various sources. This phase provided insights into the framework's scalability, operational efficiency, and resource management when dealing with a substantial and diverse corpus of smart contract code. All experimentation and tests were conducted on a dedicated server, boasting a robust hardware configuration with 16 gigabytes of RAM and 6 central processing units (CPUs) of 2.20 GHz.

4.4. Datasets

Prior to embarking on our experimental journey, we conducted a review of existing literature in the domain of smart contract vulnerabilities [17]. This systematic examination unveiled a gap in the availability of datasets that encompassed a specific class of vulnerabilities, namely signature malleability, code with no effect, and hash collision vulnerabilities. Despite the plethora of research efforts in the field of smart contracts, there was a conspicuous absence of datasets that comprehensively covered these vulnerabilities.

In response to the identified gap in existing datasets, we initiated the first phase of our experimental study. This phase entailed the deliberate creation of nine distinct smart contracts [31], each crafted to exemplify one of the aforementioned vulnerabilities. Our intention was to establish a diverse set of vulnerable scenarios, encompassing signature malleability, code segments with minimal impact, and instances of hash collision vulnerabilities.

To ensure the effectiveness of our experimentation, we employed a rigorous approach in the design and implementation of these vulnerable smart contracts. Each contract was carefully constructed to encapsulate the specific vulnerability under investigation. In the case of signature malleability, for instance, we deliberately crafted contracts where signature manipulation was feasible. Similarly, for code with no effect, we introduced code that had no substantive effect on contract execution. Finally, for hash collision vulnerabilities, we designed contracts with inputs that could potentially lead to collisions in cryptographic hash functions.

It is noteworthy that the creation of these nine unique vulnerable smart contracts stands as a significant contribution of this paper. These meticulously designed contracts are intended to fill the void in existing datasets and serve as a valuable resource for the research community. In particular, they are poised to play a pivotal role in guiding and facilitating future research endeavors. Researchers and scholars exploring smart contract vulnerabilities in the future can leverage this dataset as a foundational benchmark for their studies. By examining and analyzing these contracts, they can gain deeper insights into the intricacies of signature malleability, code with no effect, and hash collision vulnerabilities. Additionally, these contracts can serve as test cases for the development and validation of novel security analysis tools and techniques.

In essence, the creation of these vulnerable smart contracts not only advances the scope of this paper but also contributes to the broader landscape of smart contract security research. Their utility extends beyond the confines of this study, offering a valuable resource that has the potential to catalyze future investigations and enhance our collective understanding of smart contract vulnerabilities. In the last phase of our experimentation, an evaluation of SmartSentry's performance was undertaken, bearing paramount significance in substantiating its real-world utility. To facilitate this assessment, we curated a dataset encompassing 1,000 recently validated smart contracts. These smart contracts were sourced from Etherscan, a reputable platform for Ethereum blockchain data [32].

It is imperative to note that this dataset was systematically compiled on January 18, 2024, ensuring the inclusion of the most current and validated smart contracts available at the time of experimentation. As SmartSentry require the compilation of the scanned smart contract source code, accessing and analyzing the source code of these smart contracts was a fundamental aspect of our performance evaluation. To this end, we engineered a scraping script. By extracting the source code, we equipped ourselves to subject these smart

contracts to SmartSentry, thereby facilitating an in-depth analysis of the framework's performance under real-world conditions.

5. RESULTS AND ANALYSIS

5.1. Framework's effectiveness

In the first phase of our experimentation, we conducted an evaluation of SmartSentry. The objective was to assess the framework's efficacy in identifying and mitigating known vulnerabilities within smart contracts. Specifically, we subjected nine deliberately crafted smart contracts, each intentionally injected with various vulnerabilities, to SmartSentry. The results shown in Table 1 were highly encouraging and demonstrated the robustness of our tool.

SmartSentry proved to be exceptionally proficient in this controlled environment. It successfully compiled all nine smart contracts, and most notably, it accurately detected and flagged every injected vulnerability within these contracts. This achievement affirmed SmartSentry's effectiveness in identifying a range of known vulnerabilities, including signature malleability, code with no effect, and hash collision vulnerabilities. The tool's ability to comprehensively and reliably identify these vulnerabilities underscores its potential significance in enhancing the security of smart contracts.

The success of experimentation 1 not only validated the core concepts and methodologies underpinning SmartSentry but also instilled confidence in its capabilities. These promising results served as a strong foundation for the subsequent phases of our experimentation, where SmartSentry's performance was further evaluated in real-world scenarios. The effectiveness demonstrated in this controlled environment laid the groundwork for our pursuit of comprehensive smart contract security analysis and underlined the framework's potential contribution to the broader field of blockchain security.

Table 1. List of injected and detected vulnerabilities in the 9 crafted smart contracts

	SM		Hash collision		Code with no effect	
	Injected	Detected	Injected	Detected	Injected	Detected
SC 1					1	1
SC 2					1	1
SC 3					1	1
SC 4			1	1		
SC 5			1	1		
SC 6			3	3		
SC 7	1	1				
SC 8	1	1				
SC 9	1	1				

5.2. Overall performance

In the second phase of our comprehensive experimentation, we turned our focus towards evaluating the performance of SmartSentry. The primary objective was to assess the framework's operational efficiency and its ability to uncover vulnerabilities within real-world production smart contracts, thereby gauging its practical utility. The results obtained in experimentation 2 and shown in Table 2 were notable and provided valuable insights. Table 2 demonstrated the performance of SmartSentry by successfully compiling and scanning approximately 98.8% of the retrieved smart contracts in 29 minutes and 42 seconds. Remarkably, it identified 7 vulnerabilities, within these production-grade smart contracts, affirming it is capacity to detect real-world vulnerabilities effectively.

The 7 discovered vulnerabilities were manually checked by smart contract security auditor that confirmed the existence of the vulnerabilities. However, it is important to acknowledge that the framework did not identify vulnerabilities related to code with no effect during this phase. To check if the framework have missed any other vulnerabilities in the 1000 scanned smart contracts with a certainty of 95% we have used the Slovin's formula (1), as the distribution is unknown.

$$n = \frac{N}{1+NE^2} \quad (1)$$

Where n is number of samples, N is total number of smart contracts (1,000) and E is error tolerance (level).

Table 2. List of vulnerabilities discovered in the 1,000 scanned smart contracts

	Signature maleability	Hash collision	Code with no effect
0×3606…	1	0	0
0×0896…	1	0	0
0×1f7b…	0	2	0
0×F093…	0	2	0
0×99fA…	0	1	0

Subsequent a manual analysis of 286 randomly selected scanned smart contracts from the 1000 ones revealed a crucial insight. The majority of these contracts were constructed based on well-established, thoroughly tested smart contract templates, imparting a high degree of resilience against such vulnerabilities. Moreover, code with no effect vulnerabilities is typically detected and addressed during the development phase of protocols, rendering them exceedingly rare in live blockchain ecosystems.

Our framework's ability to detect seven vulnerabilities in live production smart contracts demonstrates its practical efficacy and significance in enhancing blockchain security. By identifying critical issues early in the development process, our tool not only mitigates potential security breaches but also saves developers considerable time and resources. This proactive approach contributes to the reliability and trustworthiness of blockchain ecosystems. Additionally, our framework's success underscores its potential for widespread industry adoption, influencing future standards and best practices in smart contract development. These advancements affirm the framework's value, making it a pivotal contribution to the field of smart contract security. In addition, its modular architecture allow for easy integration of new vulnerabilities detector.

Despit it is promising results SmartSentry has some limitation related to the necessity to compile the entire source code in order to extract and analyze the AST. Consequently, vulnerabilities can only be detected once the code is complete and compilable, potentially increasing the developers' workload. However, the extraction of AST information post-compilation effectively minimizes false positives that may arise from incomplete or erroneous ASTs. Moreover, SmartSentry detectors require an advanced Expert knowledge to be built, which make creating more detectors time consuming. However, this opens new directions for future researchs to also automate the process of integrating new detectors to reduce the required time.

6. CONCLUSIONS

The detection of vulnerabilities in smart contracts is crucial for ensuring the security and reliability of blockchain-based applications. SmartSentry addresses this critical need by identifying three previously unstudied vulnerabilities in smart contracts. Through its modular architecture, it successfully identified seven vulnerabilities in live production smart contracts, demonstrating its practical efficacy and importance. By focusing on early detection during the development lifecycle, SmartSentry mitigates security risks, ultimately enhancing the trustworthiness of blockchain ecosystems. Our results clearly show that manual auditing or using old tools alone may not uncover all vulnerabilities. Therefore, our framework provides a necessary and effective solution that significantly improves smart contract security.

In addition to SmartSentry, we have contributed to the scholarly community by creating a robust dataset of vulnerable smart contracts. This dataset comprises smart contracts deliberately injected with three distinct vulnerabilities: signature malleability, code with no effect, and hash collision vulnerabilities. These vulnerable contracts serve as a valuable resource for researchers, enabling them to explore, analyze, and develop innovative solutions to mitigate these vulnerabilities in real-world blockchain applications. This dataset is publicly accessible, ensuring its availability for future researchs.

Future research should explore the development of new detectors for other unexamined vulnerabilities, leveraging the modular nature of our framework. Additionally, automating the creation of these detectors using machine learning models could further advance the field, reducing the dependency on expert knowledge and expediting the detection process. We call upon the research community and industry practitioners to adopt and expand upon our framework, driving forward the collective effort to secure blockchain technologies.

REFERENCES

[1] V. Buterin, "A next generation smart contract and decentralized application platform." pp. 1–36.
[2] W. Zou *et al.*, "Smart contract development: Challenges and opportunities," *IEEE transactions on software engineering*, vol. 47, no. 10, pp. 2084–2106, 2019.

[3] O. Zaazaa and H. El Bakkali, "Unveiling the landscape of smart contract vulnerabilities: a detailed examination and codification of vulnerabilities in prominent blockchains," *International Journal of Computer Networks and Communications*, vol. 15, no. 6, pp. 55–75, 2023, doi: 10.5121/ijcnc.2023.15603.
[4] C. Shier *et al.*, "Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack," *SSRN Electronic Journal*, 2017, doi: 10.2139/ssrn.3014782.
[5] J. Gao, H. Liu, C. Liu, Q. Li, Z. Guan, and Z. Chen, "EASYFLOW: keep ethereum away from overflow," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, May 2019, pp. 23–26, doi: 10.1109/ICSE-Companion.2019.00029.
[6] A. Austin, C. Holmgreen, and L. Williams, "A comparison of the efficiency and effectiveness of vulnerability discovery techniques," *Information and Software Technology*, vol. 55, no. 7, pp. 1279–1288, 2013, doi: 10.1016/j.infsof.2012.11.007.
[7] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "eThor: practical and provably sound static analysis of ethereum smart contracts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, Oct. 2020, pp. 621–640, doi: 10.1145/3372297.3417250.
[8] T. D. Nguyen, L. H. Pham, and J. Sun, "SGUARD: towards fixing vulnerable smart contracts automatically," in *2021 IEEE Symposium on Security and Privacy (SP)*, May 2021, pp. 1215–1229, doi: 10.1109/SP40001.2021.00057.
[9] D. Wang, B. Jiang, and W. K. Chan, "WANA: symbolic execution of wasm bytecode for cross-platform smart contract vulnerability detection," *arXiv*, 2020.
[10] X. Hao, W. Ren, W. Zheng, and T. Zhu, "SCScan: A SVM-based scanning system for vulnerabilities in blockchain smart contracts," in *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, Dec. 2020, pp. 1598–1605, doi: 10.1109/TrustCom50675.2020.00221.
[11] J. Ye, M. Ma, Y. Lin, Y. Sui, and Y. Xue, "Clairvoyance," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*, Jun. 2020, pp. 274–275, doi: 10.1145/3377812.3390908.
[12] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, May 2019, pp. 8–15, doi: 10.1109/WETSEB.2019.00008.
[13] Y. X. Tang, Z. H. Li, and Y. X. Bai, "Rethinking of reentrancy on the Ethereum," in *Proceedings-2021 IEEE International Conference on Dependable, Autonomic and Secure Computing, International Conference on Pervasive Intelligence and Computing, International Conference on Cloud and Big Data Computing and International Conference on Cyber*, 2021, pp. 68–75, doi: 10.1109/DASC-PICom-CBDCom-CyberSciTech52372.2021.00025.
[14] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "MadMax: surviving out-of-gas conditions in Ethereum smart contracts," in *Proceedings of the ACM on Programming Languages*, Oct. 2018, vol. 2, pp. 1–27, doi: 10.1145/3276486.
[15] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sFuzz," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, Jun. 2020, pp. 778–788, doi: 10.1145/3377811.3380334.
[16] M. Ren *et al.*, "SCStudio: a secure and efficient integrated development environment for smart contracts," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Jul. 2021, pp. 666–669, doi: 10.1145/3460319.3469078.
[17] O. Zaazaa and H. El Bakkali, "A systematic literature review of undiscovered vulnerabilities and tools in smart contract technology," *Journal of Intelligent Systems*, vol. 32, no. 1, 2023, doi: 10.1515/jisys-2023-0038.
[18] X. Li, J. Yang, J. Chen, Y. Tang, and X. Gao, "Characterizing Ethereum upgradable smart contracts and their security implications," *arXiv preprint arXiv:2403.01290*, Mar. 2024.
[19] "Ethereum daily deployed contracts chart," *Etherscan*. https://etherscan.io/chart/deployed-contracts (accessed Jul. 22, 2024).
[20] O. Safaryan *et al.*, "Modern hash collision cyberattacks and methods of their detection and neutralization," *Journal of Physics: Conference Series*, vol. 2131, no. 2, 2021, doi: 10.1088/1742-6596/2131/2/022099.
[21] O. Zaazaa and H. El Bakkali, "Dynamic vulnerability detection approaches and tools: State of the Art," in *2020 Fourth International Conference On Intelligent Computing in Data Sciences (ICDS)*, Oct. 2020, pp. 1–6, doi: 10.1109/ICDS50568.2020.9268686.
[22] C. F. Torres, M. Baden, R. Norvill, B. B. Fiz Pontiveros, H. Jonker, and S. Mauw, "ÆGIS: shielding vulnerable smart contracts against attacks," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, Oct. 2020, pp. 584–597, doi: 10.1145/3320269.3384756.
[23] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "ReGuard," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, May 2018, pp. 65–68, doi: 10.1145/3183440.3183495.
[24] O. Zaazaa and H. El Bakkali, "Automatic static vulnerability detection approaches and tools: state of the art," 2022, pp. 449–459, doi: 10.1007/978-3-030-91738-8_41.
[25] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "SmartCheck," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, May 2018, pp. 9–16, doi: 10.1145/3194113.3194115.
[26] L. Luu, D. H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 254–269, 2016, doi: 10.1145/2976749.2978309.
[27] "Signature malleability detector.ts," *Gist*. https://gist.github.com/zoualid/67864a730fd8d1d1b34f7f0a75286cb6 (accessed Jul. 22, 2024).
[28] "Hash collision with multiple variables of varying Lengths.ts," *Gist*. https://gist.github.com/zoualid/fcbe1c2c6ad0390c99be46b03abe65e2 (accessed Jul. 22, 2024).
[29] "Code with no effect.ts," *Gist*. https://gist.github.com/zoualid/c6ac7ca2879e678257677c7f02f6a2ab (accessed Jul. 22, 2024).
[30] "Zoualid/datasets." https://github.com/zoualid/datasets/blob/main/ListOf1000VerifiedSmartContracts.txt (accessed Jan. 19, 2024).
[31] "VulnerableSmartContracts," *zoualid*. https://github.com/zoualid/VulnerableSmartContracts (accessed Jan. 19, 2024).
[32] "Verified contracts," *etherscan.io*. https://etherscan.io/contractsVerified (accessed Jan. 18, 2024).

BIOGRAPHIES OF AUTHORS

Oualid Zaazaa is a Ph.D. student at Ph.D. candidate at the National Higher School of Computer Science and Systems Analysis (ENSIAS), specializing in automated smart contract vulnerability detection, he stands at the forefront of cutting-edge research in the field. With an academic foundation rooted in computer science and a profound interest in cybersecurity, he is dedicated to unraveling the complexities of smart contract security. His research focus on developing innovative methodologies and tools to identify vulnerabilities within smart contracts. He can be contacted at email: oualid_zaazaa@um5.ac.ma.

Hanan El Bakkali a distinguished Senior Lecturer and Coordinator of the Information Systems Security (ISS) Program, holds a Ph.D. in Computer Science from EMI, earned in 2002. With a diverse academic background complemented by a 2015 certification as a Certified Ethical Hacker (CEH), he has established expertise in various facets of cybersecurity. Their research portfolio encompasses trust and reputation management systems, access control models, collaborative systems' resilience, privacy protection in cloud/big data realms, and leveraging artificial intelligence for security and application immunity. She can be contacted at email: hanan.elbakkali@ensias.um5.ac.ma.