UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

LUCRARE DE DISERTAȚIE

Streamlining Blockchain security audits through desktop applications

propusă de

Marius Enache-Stratulat

Sesiunea: Iulie, 2025

Coordonator științific

Conf. Dr. Andrei Arusoaie

UNIVERSITATEA ”ALEXANDRU-IOAN CUZA” DIN IAȘI

FACULTATEA DE INFORMATICĂ

Streamlining Blockchain security audits through desktop applications

Marius Enache-Stratulat

Sesiunea: Iulie, 2025

Coordonator științific

Conf. Dr. Andrei Arusoaie

Avizat,

Îndrumător lucrare de disertație,

Conf. Dr. Andrei Arusoaie.

Data: ..... Semnătura: .....

Declarație privind originalitatea conținutului lucrării de licență

Subsemnatul **Enache-Stratulat Marius** domiciliat în **România, jud. Iași, mun. Iași, strada Amurgului, nr. 10, bl. 258B, et. 4, ap. 14**, născut la data de **31 mai 2001**, identificat prin CNP **5010531046224**, absolvent al Facultății de informatică, **Facultatea de informatică** specializarea **Securitatea Informatiei**, promoția 2025, declar pe propria răspundere cunoscând consecințele falsului în declarații în sensul art. 326 din Noul Cod Penal și dispozițiile Legii Educației Naționale nr. 1/2011 art. 143 al. 4 și 5 referitoare la plagiat, că lucrarea de licență cu titlul **Streamlining Blockchain security audits through desktop applications** elaborată sub îndrumarea domnului **Conf. Dr. Andrei Arusoaie**, pe care urmează să o susțin în fața comisiei este originală, îmi aparține și îmi asum conținutul său în întregime.

De asemenea, declar că sunt de acord ca lucrarea mea de licență să fie verificată prin orice modalitate legală pentru confirmarea originalității, consimțind inclusiv la introducerea conținutului ei într-o bază de date în acest scop.

Am luat la cunoștință despre faptul că este interzisă comercializarea de lucrări științifice în vederea facilitării falsificării de către cumpărător a calității de autor al unei lucrări de licență, de diplomă sau de disertație și în acest sens, declar pe proprie răspundere că lucrarea de față nu a fost copiată ci reprezintă rodul cercetării pe care am întreprins-o.

Data: ..... Semnătura: .....

Declarație de consimțământ

Prin prezenta declar că sunt de acord ca lucrarea de licență cu titlul **Streamlining Blockchain security audits through desktop applications**, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test, etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de informatică.

De asemenea, sunt de acord ca Facultatea de informatică de la Universitatea ”Alexandru-Ioan Cuza” din Iași, să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Absolvent **Marius Enache-Stratulat**

Data: ..... Semnătura: .....

Contents

Motivation

The Blockchain Technology is relatively new if we consider the release of the Bitcoin cryptocurrency blockchain newtork, which took place in 2009. However, its foundation was formed decades ago, in 1989, when Leslie Lamport developed the Paxos protocol, a distributed consensus mechanism that became the base of many decentralized systems' implementation.

While it may be primarily associated to cryptocurrencies, blockchain has grown broader in terms of its applications. Its infrastructure's main qualities are the decentralized trust, transparency and immutability, which make it perfect in domains such as electronic voting, digital identity or finance. However, one of its advantages is also a critical shortcoming, because, once deployed, a smart contract cannot be modified, leaving the door open to attackers if there is any vulnerability to exploit, which is almost impossible to avoid completely.

This thesis' motivation is to address the most vulnerable sector of the blockchain community, the begginer developers and students. We propose a desktop application that integrates Slither, a fast and powerful security analysis tool, and GitHub's REST API to automatically scan smart contracts and provide an overall security report and a visual guide towards the vulnerabilities. This application should act as a training wheel for the begginer developers, but it can also be paired with a thorough manual analysis for the more advanced ones.

We hope that this kind of tools will improve the security and efficiency of smart contracts for the long term, which will prevent unnecessary electricity consumption and funds losses. While no security analysis tool is perfect and can miss certain vulnerabilities, it can prevent obvious and high-impact attacks from taking place and alleviate the developers' pressure to deliver secure smart contracts.

Abstract

The Blockchain, despite its popularity and presence on the internet and in the financial sector, is a relatively new technology and this fact is proved by the large amount of attacks that took place on such networks. This situation raises the need for secure and robust networks that can keep the clients' financial assets safe. Fortunately, there are various solutions to this category of problems, like Slither or Manticore, which take different approaches towards finding vulnerabilities. The aim of our paper is to highlight the strenghts of Slither as a static analysis tool and present a solution that integrates it in a streamlined manner and eases the process of scanning blockchain projects for beginner and intermediate developers.

Introduction

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Nunc mattis enim ut tellus elementum sagittis vitae et. Placerat in egestas erat imperdiet sed euismod. Urna id volutpat lacus laoreet non curabitur gravida. Blandit turpis cursus in hac habitasse platea. Eget nunc lobortis mattis aliquam faucibus. Est pellentesque elit ullamcorper dignissim cras tincidunt lobortis feugiat. Viverra maecenas accumsan lacus vel facilisis volutpat est. Non odio euismod lacinia at quis risus sed vulputate odio. Consequat ac felis donec et odio pellentesque diam volutpat commodo. Etiam sit amet nisl purus in. Tortor condimentum lacinia quis vel eros donec. Phasellus egestas tellus rutrum tellus pellentesque eu tincidunt. Aliquam id diam maecenas ultricies mi eget mauris pharetra. Enim eu turpis egestas pretium.

Chapter 1

Threats of Blockchain

The Blockchain [15] is a technology used in various applications, such as cryptocurrencies, Internet of Things, electronic voting and many others. Its transparent, fully distributed, peer to peer and append-only nature are its most notable strengths. The transactions, which may have a different meaning depending on the application they are used for, are publicly visible and cannot be modified after they are published. This means the users may verify any transaction with no need of a centralized authority.

Bitcoin, the most popular cryptocurrency at this moment, takes advantage of these characteristics, creating a decentralized financial system. Unfortunately, there are also significant shortcomings in terms of security. One of the most notorious instances where these vulnerabilities had been exploited is known as "The DAO" [15], where in 2016, an unknown attacker managed to drain $50 million USD through a *Reentrancy Attack*.

This thesis will first present the variety of attacks that blockchain networks are vulnerable to, their manner of work and potential effects to its victims. The Quadriga Initiative [10] - a community-based platform - hosts a list of most of the attacks and frauds that involve cryptocurrency exchanges. Among the most well-known attacks, the database of case studies includes instances of Reentrancy Attacks, Replay Attacks and Short Address Attacks.

Reentrancy Attacks [6] can take place whenever the developers of the targeted network do not update the balances before sending the funds. In this case, an attacker could recursively call the withdraw() method and drain all the available funds.

Replay Attacks [17] are specific to the case where a cryptocurrency is forked into

two separate currencies. In such a scenario, it is possible to sniff a regular transaction from any of the chains and "replay" it on the other. Processing both transaction causes the user to lose the same amount of assets twice (for example, paying two ether instead of one for the same product or service). This attack was easily possible in the Ethereum blockchain before the implementation of chainID in transactions.

Short Address Attacks [15] represent the exploit of a bug in the EVM – Ethereum Virtual Machine – used to obtain extra tokens on purchases. This category of attacks requires creating an Ethereum wallet with its address ending with a 0 byte. The attacker makes a purchase on the address by removing the last digit and causes the contract to try to append the missing byte to the incomplete address, but it ends up appending it to the paid amount. In this case, the contract pays 256 times more tokens than intended.

In terms of prevention and detection of such attacks, there are numerous tools that serve this purpose. As Kaixuan Li et al. describe in their article [12], a significant category of such tools detect vulnerabilities in the code through either static analysis or symbolic execution.

Technology	Tool	Analysis Level	Stars
Static Analysis	Securify2	Source Code	529
	Slither		4 500
	SmartCheck		315
Symbolic Execution	Manticore	Bytecode	3 500
	Osiris		50
	Oyente		1 300

Table 1.1: List of the main static analysis and symbolic execution tools, adapted from Kaixuan Li et al. [12]

According to table 1.1, the static analysis tools, primarly represented by Slither, enjoy a higher popularity than the symbolic execution tools, represented by Manticore. This fact is shown by the number of stars on GitHub, which indicate how popular a tool is. These types of tools share the principle of scanning the code without executing it, but do so in different ways.

Generally, static analysis tools [5] are automatic methods that determine the runtime properties of a program without running it. In Slither's [8] case, the analysis' purpose is to detect security vulnerabilities within the smart contract and to point out

potential optimizations, which result in a lower gas consumption, an essential aspect in this field. As such, this category of security tools should be considered the opposite of Dynamic Analysis tools, which rely on running the target program and can also monitor it's performance among other aspects.

On the other hand, symbolic execution tools [1] take a similar approach of analyzing the given program without running the code, based solely on its bytecode. Esentially, Manticore [13] uses the bytecode of a smart contract to map out all the execution paths of the program. Afterwards, all the paths get explored and the tool determines what kind of inputs determine each possible outcome. Two of the main components of Manticore are the Core Engine and the Event System. The Core Engine is responsible with operating and managing the state of a program at a certain point in the execution. The Execution Module has the purpose to emulate a system that runs the target program, with a CPU, system memory and an operating system like Linux.

```
int foobar(int a, int b) {
    int x = 1, y = 0;
    if (a != 0) {
        y = 3 + x;
        if (b == 0) {
            x = 2 * (a + b);
        }
    }
    assert(x - y != 0);
}
```

Figure 1.1: Example of code to analyze. From Roberto B et al. [1]

The figure above shows a simple example of function that a symbolic execution tool can operate on. A tool like Manticore can trace all the possible paths the execution may go at run-time. The assertion at the end is the possible point of failure for this function, in case `x` and `y` are equal.

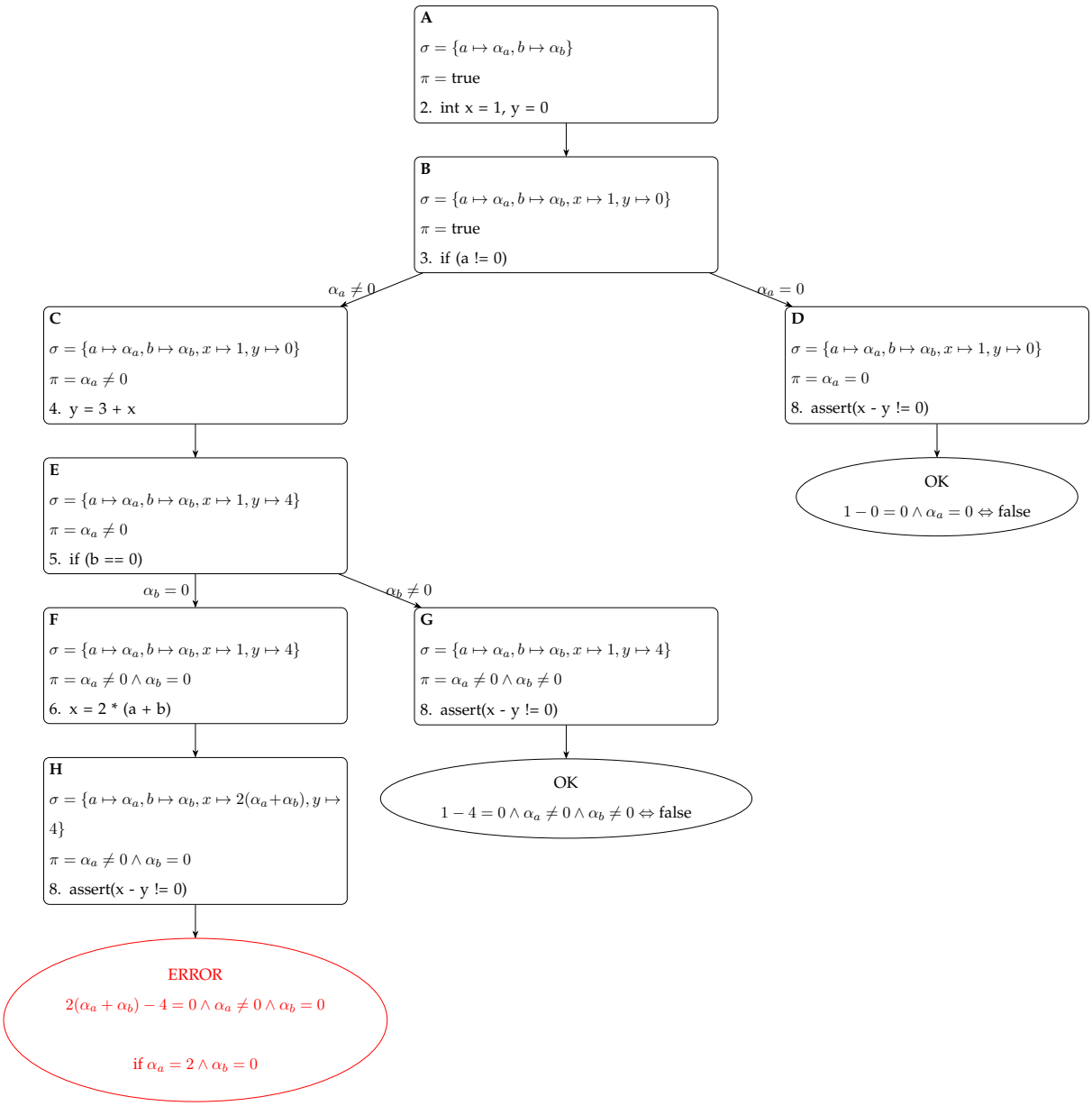

Figure 1.2: Example of symbolic execution tree of function `foobar`. Each state shows the symbolic store σ, the path constraint π, and the current statement. From Roberto B. et al [1]

As shown in this graph, σ contains all the symbols stored in the context of the function, π is a statement that represents the path constraint and below it is the line of code that gets excecuted at a given time. State *B* is the first one to fork in two hypothetical states, based on the value of the first parameter, as shown by π in the states *C* and `D`. If `a == 0`, the function skips to the final assert and returns with no failure. Otherwise, the program would continue with state `E` and the following ones. Finally, a symbolic execution tool can reveal that the function will throw an error in case `a == 2 and b == 0`.

Tool	# Success	# Failure		
		# Timeout	# Compilation	# Total
Slither	767 (97.34%)	2	19	21
Manticore	112 (14.21%)	626	50	676

Table 1.2: Success rate and failures summary of Slither and Manticore. Adapted from Kaixuan Li et al. [12]

Finally, to address Manticore's performance, as shown above, its exhaustive approach causes it to fail most of the tests due to timeout. Even if we were to ignore the timeout fails, Slither is visibly more consistent, having less compilation failures. The main flaw and reason of Manticore's high timeout rate may be the fact that it has to process large amounts of possible paths in order to find the ones where a security flaw is exploited. As such, Slither is the more consistent and better performing in comparison to Manticore.

1.1 Slither

Considering the performance metrics discussed in the first section, this section will be dedicated to Slither [8], a static analysis tool catered to smart contracts written in Solidity and designed to offer granular information about smart contract code. This tool specializes in vulnerability detection, but also manages to point out potential optimizations that lower the gas consumption of a smart contract. In order to facilitate the usage of this tool, Slither leverages printers to help the user better understand the contract's functionality and structure. This is done through representations such as the control flow graph or the inheritance graph or summaries of the contract and the issues detected in its code.

In opposition to dynamic analysis [3] tools, which rely on executing the contract's code to find security weaknesses, static analysis tools detect vulnerabilities without running the smart contract. Such approaches include parsing and decompiling the EVM bytecode and translating it into semantic facts. These semantic facts are then matched with predefined patterns meant to reveal common issues.

Slither [8] proceeds in a similar manner by taking the Solidity Abstract Syntax Tree generated by the compiler and extracting information such as the control flow graph and transforms the code given into SlithIR, its internal representation language.

After these two initialization stages, the actual code analysis takes place. In this stage, Slither identifies the reads and writes of variables, analyses the data dependency and determines whether the methods are protected from unauthorized use (for instance, only the owner of a contract may create new tokens).

This tool [8] helps auditors detect various vulnerabilities and potential security threats, such as reentrancy attacks or uninitialized storage variables. Its aproach of quickly transalating and analyzing large codebases makes it significantly faster than symbolic execution engines, as shown Table 1.2, a great feature for fast development workflows.

Furthermore, its set of detectors are able to find obvious and subtle vulnerabilities alike while providing useful educational resources to help developers fix and prevent future attacks. All these factors make Slither a great solution for quick and automated daily audit processes, preventing many critical security vulnerabilities.

Another essential topic in blockchain development, beside security concerns, is optimization, the techniques used to lower the "gas" consumption. Slither includes a set of detectors specialized in finding ways to optimize code [7] from different perspectives. They are able to find issues such as dead code, as in methods or code branches that are never called or executed, unused state variables or variables that should be declared constant to save gas.

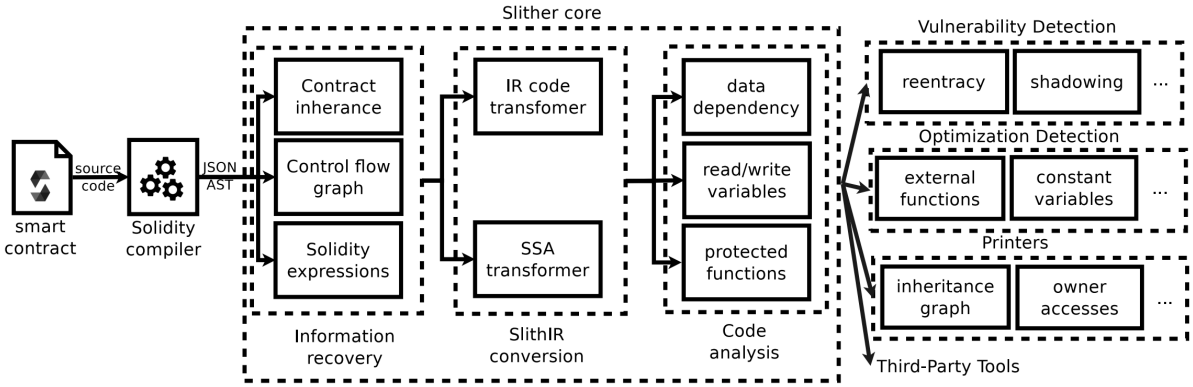

Figure 1.3: Slither overview. From Josselin F. et al. [8]

As to how it works [8], Slither uses a multi stage procedure to parse and process the codebase of a blockchain project. First, it uses the Solidity compiler to generate the Abstract Syntax Tree of the contract, from which it recovers important information: inheritance graph, control flow graph and the list of expressions. Next, the contracts get translated to SlithIR, the internal representation language. The following step is the ac-

tual code analysis, in which Slither processes data dependencies, read/write variables and protected functions to output detected vulnerabilies and optimization opportunities.

Chapter 2

Slither's performance analysis

This chapter will be dedicated to assessing Slither's performance in terms of vulnerability detection. The metrics used to determine how well security scanning tools work are: detection rate (how often a tool can correctly flag a vulnerability), analysis speed and attacks coverage (the variety of attacks that a tool can prevent).

		Slither	Securify	SmartCheck	Solhint
Accuracy	False Positive Rate	10.9%	25%	73.6%	91.3%
	Flagged contracts	112	8	793	81
	Detections per contract	3.17	2.12	10.22	2.16
Performance	Execution time(s)	0.79 ± 1	41.4 ± 46.3	10.9 ± 7.14	0.95 ± 0.35
	Time out rate	0%	20.4%	4%	0%
Robustness	Failure rate	0.1%	11.2%	10.22%	1.2%
Reentrancy examples	DAO	Yes	No	Yes	No
	Spankchain	Yes	No	No	No

Table 2.1: Performance comparison between Slither, Securify, SmartCheck and Solhint. From Josselin F. et al. [8]

The table above covers the first two performance criteria: the detection rate and analysis speed. The experiment was done by analyzing 1000 contracts focusing only on reentrancy detectors. Slither's results are visibly better than the ones obtained by its counterparts. Firstly, it's false positive rate is far lower, at just 10.9%, while Securify falls in the second place with 25%. Secondly, it's average execution time is also significantly lower than those of Securify and SmartCheck. In short, Slither is by far faster, more accurate and more consistent than the other three tools. Another important aspect is the time out rate of 0% for Slither and Solhint, which further supports Slither's

consistency in the detection of reentrancy attacks. Also, while this results do not conclude Slither's attack coverage, we can see it is the only tool that managed to detect attacks similar to the ones that targeted DAO or Spankchain, which are notorious for the amount of funds stolen and its effect on people's perception towards blockchain's security.

Vulnerabilities	Read	TP	TN	FP	FN
Re-entrancy	29/31	28	0	0	1
Access Control	18/18	14	0	0	3
Arithmetic	15/15	0	0	5	0
Unchecked LLC	26/26	26	0	0	3
Total	88	68	0	5	7

Table 2.2: Slither's coverage of security attacks. Adapted from Senan B. [2]

This table shows how Slither can detect four types of vulnerabilities: Reentrancy, Access Control, Arithmetic and Unchecked LLC and covers 4 output categories: True Positive (TP), True Negative (TN), False Positive (FP) and False Negative (FN). Out of the 90 test cases given, 88 passed the reading phase, 68 had got correctly flagged as vulnerable, five ended up as False Positives and seven as False Negatives. That equates to a rate of 77.27% for True Positive, 5.68% for False Positive and 7.95% for False Negative. Therefore, while in almost 6% of cases the user is falsely lead to believe a section of his code is vulnerable, a case that has no impact on the project's security, the rate at which Slither misses a vulnerability is almost 8%.

Vulnerabilities	Slither	SmartCheck
Bad Randomness	2.99s	8.9s
Access Control	2.98s	10s
Reentrancy	2.48s	*32.2s
Arithmetic	3.09s	21.16s

Table 2.3: Detection rate of Slither compared to SmartCheck, another static analysis tool. *SmartCheck registered a significantly lower detection rate for reentrancy attacks. Adapted from [11]

According to the table above, Slither is by far faster than SmartCheck at detecting vulnerabilities across all four types included. Also, both tools properly detected the

vulnerabilities, so the quality of Slither's detection is just as high as SmartCheck's, except for reentrancy attacks, where SmartCheck missed a significant part of the issues. In short, Slither is much faster and more reliable than SmartCheck.

Among the included attacks, there are two new types: bad randomness and arithmetic. Bad randomness [14] attacks may take place when a contract generates random numbers through unsafe methods, such as using the block's timestamp as the seed for essential operations, like asset transfer. If the attacker is able to predict the seed (for example, by initiating a transaction at a certain time), they may get an undeserved advantage over honest users and could obtain more funds, depending on the attacked application.

Arithmetic attacks [16] are specific to smart cotracts that use versions of Solidity older than Solidity 0.8. This type of attack is done by overflowing or underflowing an variable. Overflowing a variable is done by incrementing it over its limit (for an uint8 variable, adding one to 255 will return 0). Underflowing is done similarly through decrementation. Fortunately, newer versions of Solidity throw errors in these cases and the older ones have access to the SafeMath library, so these vulnerabilities are easy to solve.

As seen in the previous paragraphs, Slither is a great tool for detecting security vulnerabilities, consistently finding issues almost regardless of their category. This high detection rate is backed up by its high speed of processing smart contracts, which is another strong point of Slither. Furthermore, unlike its competition, this tool is also able to detect optimization opportunities in smart contracts.

Smart contract optimization, while not a major security concern, used to be an important subject for Ethereum-based smart contracts developers. This was the case before The Merge, when Ethereum tranzitioned from the Proof-of-Work consensus to a Proof-of-Stake one. The Proof-of-Work model relied on validators using large amounts of electricity through mining in order to verify transactions and an inefficient smart contract on top of that would only have worsened the problem. However, the shift towards the Proof-of-Stake consensus meant that Ethereum would dramatically lower it's energy consumption by over 99.9% on september 14th 2022 [4].

Now, because the energy consumption is not as threatening, optimizing a smart contract is more of a matter of reducing the risk of intruders using inefficient code as backdoors for their attacks and ensuring a fast and responsive application for its users. As far as the latest researches go, Slither [8] had been tested for detecting constant

variables that were not declared accordingly. A properly declared constant variable will take no space in the storage of the contract and can be accessed with fewer instructions when needed. However, a non-constant variable that acts as a constant (it doesn't change its value under any circumstance) could unnecessarily increase the code size and the usage cost. Slither was tested on a total of 36.000 contracts and up to 56% of them were found to have variables that acted as constants or variables that ended up not being used at all. These findings prove that Slither is able to detect basic, but useful, code optimization.

Thanks to its optimization detection and coverage of various categories of attacks, Slither is a great tool for security audits. However, it can also be used for educational purposes in blockchain related courses, where the students could run it on their projects and see its security vulnerabilities. The usage of Slither in such early stages of learning smart contract development raises awareness of the most common attacks and helps the student avoid leaving the project vulnerable in the future.

Another subject of great importance when discussing Slither, or any similar tool, are the legal aspects and the ethical considerations of using it. The first and most important matter is the fact that, while the developers of a smart contract may run Slither on their project to find and fix its vulnerabilities, intruders may do the same to find exploitable pieces of code to focus their attack on. Of course, this is mostly possible in the case of an open-source project, unless the developers are quick enough to detect and fix all the security problems before any attacker manages to exploit them. In other words, Slither may be both the guard that points out security issues and the complice that helps the attacker find the right way to exploit the contract.

Similarly, Slither can be used by regular users on open-source smart contracts to assess whether they are safe to use and may contact the developers in case they find glaring problems. In the meantime, developers should take responsability for the security of their project and conduct manual analysis of the code on top of using automated tools. In the end, security analysis tools are far from perfect and are subject to throwing false positives or negatives. As such, the developers are the only ones responsible with avoiding security attacks to the best of their abilities.

Chapter 3

SmartScan

SmartScan is a desktop application that aims to help begginer developers evaluate their projects from a security standpoint while it can also help regular users evaluate how safe a smart contract is. The main purpose of this project is to be easy to use, but also powerful enough to provide relevant insight and guidance towards fixing the security vulnerabilities. For this purpose, SmartScan uses the GitHub REST API [9] for a quick and intuitive project cloning process and Slither [7] for the analysis of the cloned project.

The choice for the analysis tool was not difficult for multiple reasons, as conveyed by the previous chapter. The first one is in its core design: being a static analysis tool makes it great for scanning projects without executing them, which would have added complexity to the entire setup. Secondly, its balance between high performance and low execution time in terms of scanning for security threats is great for providing fast and reliable vulnerability reports.

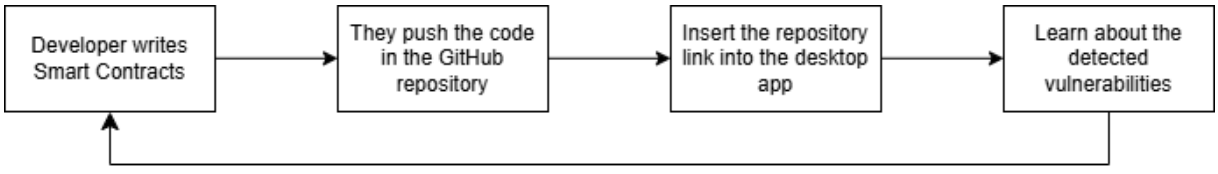

Figure 3.1: Scenario diagram for SmartScan.

As shown in the diagram above, the usual workflow in which SmartScan is involved starts with the developer writing the smart contract code and pushing it in the GitHub repository. Out application intervenes right after, when the developer copies the repository link in it and starts the analysis. Afterwards, the application returns a security report, based on which the developer may start fixing the vulnerabilities and

the cycle may repeat. Whenever the application runs the analysis, it makes sure to fetch the latest version of the project so the developer doesn't need to.

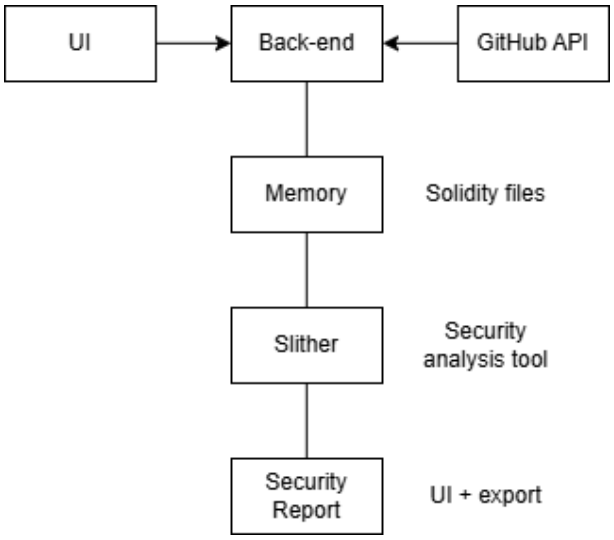

Figure 3.2: Component diagram for SmartScan.

In terms of infrastructure and design choices, SmartScan is developed in Python, using the PySide 6 framework [18] for the interface because it provides a clean and easy to use design with minimal performance limitations. The back-end of the application consists of the aforementioned REST API and Slither, linked together with custom scripts designed to manage the analysis results.

Figure 3.3: SmartScan's User Interface.

The figure above presents the user interface of SmartScan. The first component the user will interact with is the Repository Link text box, in which they will insert the repository link copied from GitHub before pressing the Analyze Button. Afterwards, the REST API will handle the cloning of the project and Slither will analyze it, returning all the found vulnerabilities. The File Tree is used to access the cloned project and open any Solidity file in order to see the code inside. The Code Area will host the code of the currently open file, with a side bar that shows the number of each line and a yellow highlight that shows which lines of code are vulnerable, as detected by Slither. The side bar and the highlight are meant to visually guide the developer towards the problem with ease. After fixing the code, the user may save the file through the dedicated button on the top left corner.

SmartScan, while using external APIs for cloning and analyzing the project, compiles a security report of its own, based on the vulnerabilities found by Slither. The whole report will be saved in a separate text file, so the user can access it whenever they want, and will be shown in the Code Area at the end of the analysis. Also, the stars next to the Save Button will turn red depending on the determined severity of the project. The metric used to measure how vulnerable a project is will be the amount of threats found and their severity. Each star turning red represents a higher vulnerability of the project as a whole. Ideally, the analyzed project should get a rating of zero stars, while a five-star rating is the worst case possible.

Star rating	Condition
$0\star$	No vulnerability found
$1\star$	≤ 10 vulnerabilities, no high severity ones
$2\star$	≤ 25 vulnerabilities, ≤ 2 of high severity, no critical ones
$3\star$	> 25 vulnerabilities, ≤ 5 of high severity, no critical ones
$4\star$	≤ 10 high vulnerabilities or ≤ 2 critical ones
$5\star$	> 10 high vulnerabilities or > 2 critical ones

Table 3.1: Project vulnerability rating done by SmartCheck.

A zero-star rating is obtained if Slither finds no security issue (informational and optimization issues are not included). A One-star rating is awarded if Slither finds less than 10 vulnerabilities, none of which are of high or critical severity. For a project to get a three-star rating, the project is required to have no critical vulnerability, but is

allowed to have up to five high severity vulnerabilities. If any critical vulnerability is found, the project will get a rating of four or five stars regardless of how many issues are found.

The idea behind this rating is to raise awareness to the developer towards the security threats and induce a sense of urgency towards solving the high and critical vulnerabilities. For this reason, any critical vulnerability found will grant a minimum of four stars, while projects with no high severity vulnerabilities will only get one star. Furthermode, while one could argue that the rating should factor in the project's complexity and allow more vulnerabilities for larger projects, we should also consider that high-scale projects may also get targeted more often by attackers while the financial stakes are also higher. As such, the state of a project's security should not be influenced by its size.

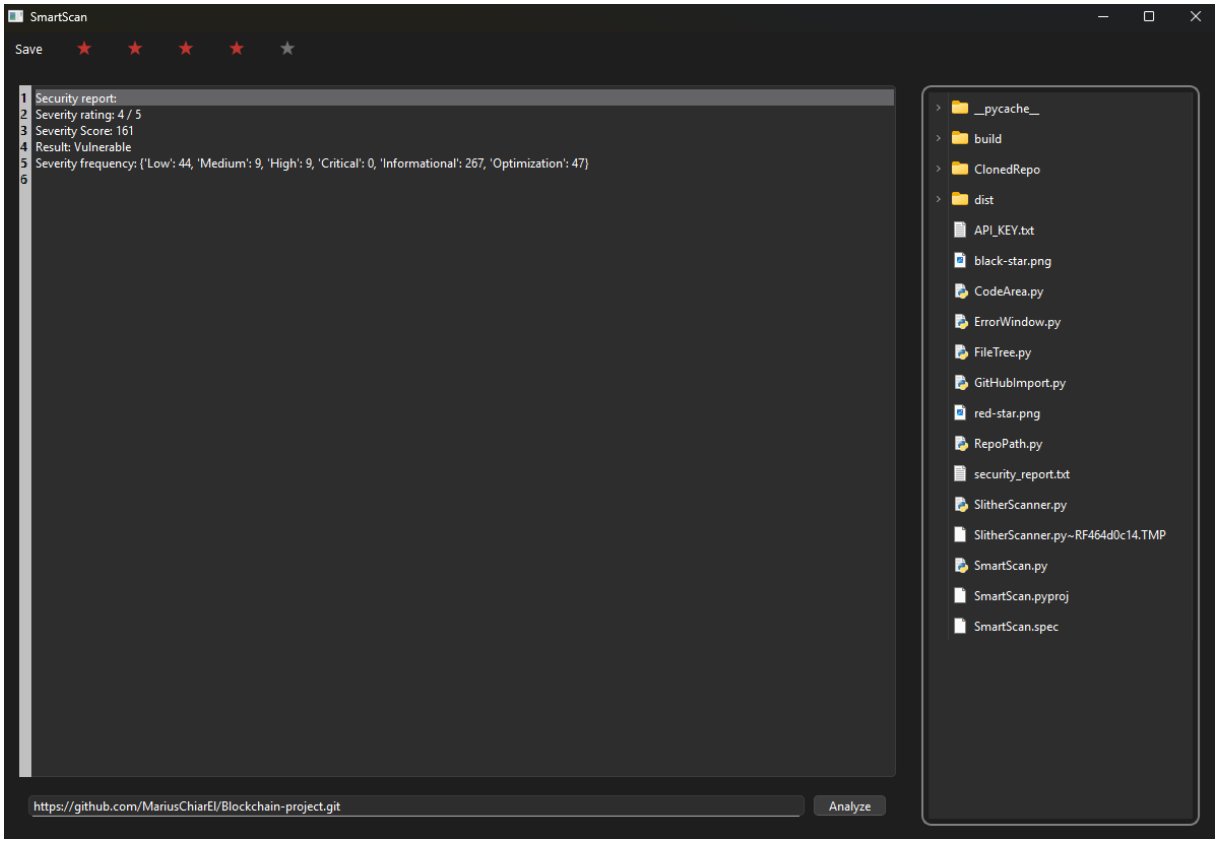

Figure 3.4: SmartScan's User Interface in practice: The Code Area features the final security report of a smart contract.

As shown in the screenshot above, our sample smart contract project obtained a rating of four stars. Despite having no critical security issue, the high severity vulnerabilities are the root cause of these results. In this project, Slither had found 44 low severity errors, nine of medium severity and another nine of high severity. Addition-

ally, the tool found 47 opportunities to optimize the smart contract and 267 informational errors, which are often not indicating any direct problems for the end user.

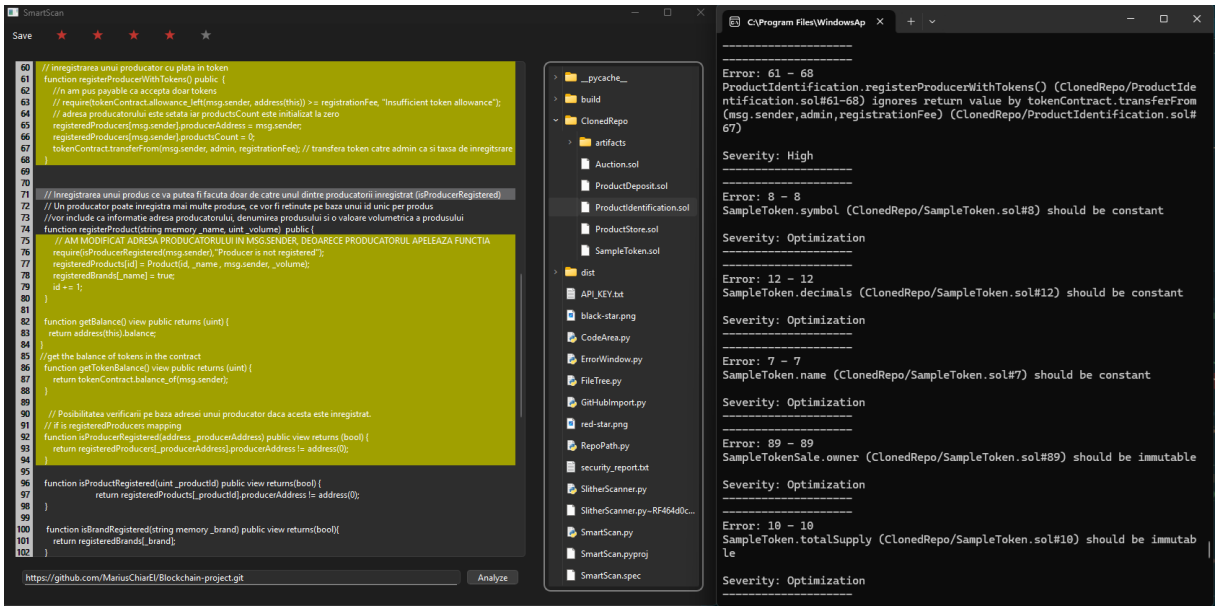

Figure 3.5: SmartScan's User Interface in practice: The Code Area features the code of a Solidity file with its affected lines highlighted. The console shows the description of each error.

The figure above shows SmartScan in action, highlighting the errors, as reported in the console. For instance, we can see the `registerProducerWithTokens()` method, containing an error at line 67, where `transferFrom()` [19] was called. Slither detected that this function had been called without checking its result, as it returns a boolean that tells whether the transfer succeeded. In this case, the registration proceed while the user does not have the funds to do so. This vulnerability has a high severity because any user can obtain the producer title without paying the smart contract as intended. This kind of vulnerability has the highest impact on the smart contract owner because they are the one to not get the funds they should have.

Concluzii

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Nunc mattis enim ut tellus elementum sagittis vitae et. Placerat in egestas erat imperdiet sed euismod. Urna id volutpat lacus laoreet non curabitur gravida. Blandit turpis cursus in hac habitasse platea. Eget nunc lobortis mattis aliquam faucibus. Est pellentesque elit ullamcorper dignissim cras tincidunt lobortis feugiat. Viverra maecenas accumsan lacus vel facilisis volutpat est. Non odio euismod lacinia at quis risus sed vulputate odio. Consequat ac felis donec et odio pellentesque diam volutpat commodo. Etiam sit amet nisl purus in. Tortor condimentum lacinia quis vel eros donec. Phasellus egestas tellus rutrum tellus pellentesque eu tincidunt. Aliquam id diam maecenas ultricies mi eget mauris pharetra. Enim eu turpis egestas pretium. **[?]**

Bibliography

[1] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys*, 2018.

[2] Senan Behan. Solidity smart contract testing with static analysis tools. *Blockchain Research and Applications*, 2022.

[3] Lyubka Dencheva. Comparative analysis of static application security testing (sast) and dynamic application security testing (dast) by using open-source web application penetration testing tools. *International Journal of Computer Applications*, 2022.

[4] Digiconomist. Ethereum energy consumption index. `https://digiconomist.net/ethereum-energy-consumption`, 2023. Accessed: 2025-05-22.

[5] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science*, 2008.

[6] Noama F. and Manar H. Reentrancy vulnerability identification in ethereum smart contracts. *International Journal of Advanced Computer Science and Applications*, 2021.

[7] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither github repository. `https://github.com/crytic/slither`, 2018.

[8] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. *arXiv preprint arXiv:1908.09834*, 2019.

[9] GitHub. Github rest api documentation. `https://docs.github.com/en/rest?apiVersion=2022-11-28`, 2025. Accessed: 2025-06-08.

[10] Quadriga Initiative. Quadriga initiative. https://quadrigainitiative.com/indexjay.php. Accessed: 2025-05-24.

[11] Bahareh Lashkari and Petr Musilek. Evaluation of smart contract vulnerability analysis tools: A domain-specific perspective. *IEEE Access*, 2023.

[12] Kaixuan Li et al. Static application security testing (sast) tools for smart contracts: How far are we? *IEEE Access*, 2024.

[13] Mark Mo, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. *arXiv preprint arXiv:1907.03890*, 2019.

[14] Peng Qian, Jianting He, Lingling Lu, Siwei Wu, Zhipeng Lu, Lei Wu, Yajin Zhou, and Qinming He. Demystifying random number in ethereum smart contract: Taxonomy, vulnerability identification, and attack detection. *IEEE Transactions on Information Forensics and Security*, 2023.

[15] Muhammad Saad, Jeffrey Spaulding, Laurent Njilla, Charles Kamhoua, Sachin Shetty, DaeHun Nyang, and Aziz Mohaisen. Exploring the attack surface of blockchain: A systematic overview. *IEEE Communications Surveys & Tutorials*, 2019.

[16] SmartState. Understanding overflow and underflow vulnerabilities in smart contracts. https://smartstate.tech/blog/understanding-overflow-and-underflow-vulnerabilities-in-smart-contr html, 2023. Accessed: 2025-05-22.

[17] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[18] The Qt Company. Pyside6 documentation. https://doc.qt.io/qtforpython-6/, 2025. Accessed: 2025-06-11.

[19] The Solidity Authors. Solidity documentation. https://docs.soliditylang.org/en/v0.8.30/, 2025. Accessed: 2025-06-12.