A Static Analysis-based Platform-as-Service to Improve the Quality of Smart Contracts

Dinis Antunes Palha de Araújo

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Rui Filipe Lima Maranhão de Abreu
Prof. João Fernando Peixoto Ferreira

Examination Committee

Chairperson: David Manuel Martins de Matos
Supervisor: Prof. Rui Filipe Lima Maranhão de Abreu
Member of the Committee: Miguel Filipe Leitão Pardal

October 2021

Acknowledgments

I would like to thank my supervisors João F. Ferreira and Rui Maranhão for introducing me to this thesis' subject and for their continuous insight and support in this journey. I would also like to thank my parents for their encouragement and caring over all these years, for always being there for me and without whom this project would not be possible. Finally, I thank my family and friends for their consideration and help throughout all these years.

Abstract

Blockchain technology promises consensus on a decentralized, immutable digital ledger maintained by a peer-to-peer network involving nodes that do not trust each other. Such technology has gained both popularity and financial value. Likewise, Ethereum's Smart Contracts followed this trend, by promising distributed computations without the need for a third party. More than one and a half million smart contracts have been deployed to the Ethereum Blockchain where some have been known to have security vulnerabilities. Recent attacks on these vulnerabilities result in great financial losses and due to the smart contract immutability these cannot be patched. Many static analysis tools have been created for Smart Contracts and SmartBugs was created as an extensible framework that aggregates them with the goal of detecting different vulnerability types. While using SmartBugs, developers tend to find themselves with a huge overhead on parsing the outputs, unifying the results and acknowledging the information produced by this tool. In this dissertation we describe SARIF, a standard format for the output of static analysis tools. We then discuss how we converted the 11 tools run by SmartBugs to this format. Moreover, we develop SASP, a protocol server, as a service to receive queries, run the analysis and aggregate the results. Finally, we overview the GitHub pull based development platform as a methodology to automate the analysis and show its results in a more intuitive manner. Our goal is to increase the accessibility of static analysis tools to smart contract developers.

Keywords

Blockchain; Ethereum; Smart Contracts; Static Analysis; SmartBugs; Static Analysis Results Interchange Format (SARIF); Static Analysis Server Protocol (SASP); GitHub

Resumo

A tecnologia Blockchain garante concordância num conjunto de dados desentralizados e imutáveis entre entidades peer-to-peer sem requerer confiança entre os mesmos. Esta tecnologia tem ganhado imensa popularidade e valor financeiro nos últimos anos. Da mesma forma, os smart contracts da rede Ethereum têm seguido o mesmo rumo. Mais de um milhão e meio de smart contracts foram lançados para a rede, nos quais foram encontrados algumas vulnerabilidades sérias. Ataques recentes a explorar estas vulnerabilidades resultaram em perdas financeiras extremas e, devido à imutabilidade dos smart contracts, estas vulnerabilidades são quase impossíveis de serem reparadas. Já foram criados vários analisadores estáticos para smart contracts e o programa SmartBugs foi desenvolvido como uma ferramenta extensível que agrega estes analisares de forma a detetar o maior número de vulnerabilidades antes do contrato ser lançado. Enquanto usam o SmartBugs, os programadores tendem a encontrar-se com uma grande sobrecarga para analisar os relatórios, unificar os resultados e entender as informações providenciadas por esta ferramenta. Nesta dissertação descrevemos o SARIF, um formato padrão para a informação gerada por analisadores estáticos, e também discutimos como convertemos as 11 ferramentas do SmartBugs para este formato. Além disso, programámos o protocolo SASP como um serviço que recebe pedidos de análise, executa a análise e agrega os resultados. Também explicamos a plataforma de desenvolvimento do GitHub como uma possível maneira de automatizar a análise e mostrar os resultados de forma mais intuitiva. O nosso objetivo é a maior adoção de ferramentas de análise estática por parte dos programadores.

Palavras Chave

Blockchain; Ethereum; Smart Contracts; Static Analysis; SmartBugs; SARIF; SASP; GitHub

Contents

1 Introduction **1**

1.1 Motivation 3

1.2 Objectives and Contribution 4

1.3 Thesis Outline 5

2 Background **7**

2.1 Blockchain 9

2.2 Ethereum 11

2.2.1 Smart Contracts 11

2.3 Vulnerabilities of Ethereum Smart Contracts 12

2.4 SmartBugs 14

2.4.1 Web-based User Interface 15

2.4.2 Tools 16

2.4.3 Datasets 17

2.4.4 Analysis 18

2.5 SARIF 19

2.5.1 Standard OASIS 19

2.5.2 SARIF - A comprehensive analysis standard 20

2.5.2.A Root Structure 20

2.5.2.B Analysis Results 21

2.5.2.C Analysis Metadata 26

2.6 SASP 27

2.7 GitHub 29

2.7.1 Workflow 29

2.7.2 Actions 31

2.7.3 Marketplace 32

3 Implementation **35**

3.1 Motivation 37

3.2 SARIF Converters 38
3.2.1 SARIF's Vulnerabilities Mapping Table 43
3.3 SASP 43
3.4 GitHub Review Bot 45
3.4.1 NoSASP approach 48
3.4.2 False positives 48
3.5 SmartBugs 49
3.5.1 Conkas 49
3.5.2 Unit Tests 50
3.5.3 System Requirements 51
3.5.4 Updated CLI 51
3.5.5 Updated Methodology for adding tools 51

4 Evaluation **55**
4.1 SARIF converters 57
4.1.1 Converter Execution 57
4.1.2 SARIF Standard Validation 57
4.1.3 Vulnerabilities Verification 59
4.2 SASP Service 61
4.2.1 Functionality 61
4.2.2 Stress Test 62
4.3 GitHub Adapter 64
4.3.1 Functionality 64
4.3.2 Uniqueness 66
4.3.3 Intuitiveness 66

5 Conclusion and Future Work **69**
5.1 Achievements 71
5.2 Future Work 72

List of Figures

2.1 Proof-of-Work (PoW) Chain of Blocks (Adapted from: [1]) 11
2.2 SmartBugs Architecture (Adapted from: [2]) 15
2.3 SmartBugs' Web-based User Interface (WUI) Dashboard 16
2.4 Point-to-point connections between static analysis tools and developer tools (Adapted from: [3]). 27
2.5 SASP Vision: Using SASP for all communications between static analysis tools and developer tools (Adapted from: [3]). 28
2.6 GitHub's new branch workflow. SmartBugs automated analysis is recommended to be integrated into the review and discussion step (in purple) (Source: [4]). 31

3.1 Overview of the final implementation. This concept can be applied to any other static analysis tool. 38
3.2 SarifHolder Construction Design - A simplified SARIF version ready to be consumed . . . 40
3.3 Example of a end consumer working with SASP to populate comments for an arbitrary code submitting 44
3.4 GitHub displaying the vulnerabilities reported by Oyente on the SimpleDAO contract . . . 47
3.5 GitHub inflating the SimpleDAO contract with the Reentrancy Vulnerability reported by Oyente 47
3.6 GitHub option to report false positives 49

4.1 Upload of a correctly formatted SARIF file on the Microsoft's Validation tool (From: [5]) . . 58
4.2 Upload of a badly formatted SARIF file on the Microsoft's Validation tool (From: [5]) 58
4.3 Original output of the eth_tx_order_dependence_minimal.sol analysis by Securify 60
4.4 SARIF output of the eth_tx_order_dependence_minimal.sol analysis by Securify processed by Microsoft's SARIF Viewer tool (From: [6]) 61
4.5 SASP stress testing with Oyente analysing SimpleDAO 63
4.6 SASP stress testing with Mythril and Slither analysing SimpleDAO and Reentrancy 63

4.7 GitHub SARIF interface after consuming the output sent by the SASP service. 65
4.8 SARIF Viewer displaying the SARIF file produced by SmartBugs locally. 66
4.9 The original Slither output. 67
4.10 GitHub SARIF interface after consuming the output produced by Slither. 68

List of Tables

2.1 Categories of vulnerabilities available in the $SB^{CURATED}$ (Source: [2]) 18
2.2 Accuracy of the 7 tools based on the vulnerabilities detected on the $SB^{CURATED}$ analysis' result (Source: [2]) 19
2.3 Combined accuracy of the 7 tools based on the vulnerabilities detected on the $SB^{CURATED}$ analysis' result (Source: [2]) 19

3.1 SARIF Vulnerability Mapping Table for Oyente 43
3.2 Conkas Execution Time Statistics (Source: [7]) 50
3.3 Conkas Accuracy Metrics (Source: [7]) 50

Listings

2.1 SARIF Root Object 21

2.2 SARIF *runs* subkeys 21

2.3 SARIF *artifacts* subkeys 22

2.4 SARIF *invocations* subkeys 22

2.5 SARIF *logicalLocation* subkeys 23

2.6 SARIF *results* subkeys 24

2.7 SARIF *tool* object 26

3.1 SARIF output produced by Oyente analysing SimpleDAO 41

3.2 Example of the file *.github/workflows/smartbugs.yml* that triggers the SmartBugs static analysis automatically at every pull request 46

Acronyms

API	Application Program Interface
CI	Continuous Integration
CLI	Command-Line Interface
DASP	Decentralized Application Security Project
EVM	Ethereum Virtual Machine
ETH	Ether
IDEs	Integrated Development Environments
JSON	JavaScript Object Notation
OASIS	Organization for the Advancement of Structured Information Standards
SARIF	Static Analysis Results Interchange Format
SASP	Static Analysis Server Protocol
PaS	Platform-as-Service
PoW	Proof-of-Work
PoS	Proof-of-Stake
WUI	Web-based User Interface

1 Introduction

Contents

1.1 Motivation 3
1.2 Objectives and Contribution 4
1.3 Thesis Outline 5

1.1 Motivation

An asynchronous peer-to-peer network of untrusting nodes, spread all across the world, sharing one unique immutable digital ledger: what would have been considered science fiction prior to 2008, has become a reality through Blockchain [8]. Blockchain technology has gained massive attention since the introduction of Bitcoin's white paper in 2008 [1]. Blockchain earned its name for being a growing list of blocks where each block contains the hash of the previous one, such as they are linked to each other. So in order to switch an older block, it would be necessary to change every single subsequent block. Cryptocurrencies, which have grown in popularity around the world, use this data structure as a public ledger to keep track of all validated transactions. Blockchain is a public append-only database that retains a permanent and immutable record of digital transactions. It is secured by a peer-to-peer network utilizing a consensus process rather than relying on trust.

Particularly, in the Bitcoin implementation, each block consists of a record of digital transactions which, when verified and accepted by the network, will be a part of a global immutable distributed ledger. The network is composed by a peer-to-peer decentralized system to be able to persist when a particular node is down or attacked and to increase the system trustworthiness. In this network no node trusts each other, however due to its cryptographic properties, they will always agree with each other by maintaining a shared global ledger while overcoming the missing trust [9, 10].

One of the positive impacts of blockchain is that it is a system that allows for safe distributed computations even when third parties are not trusted. Since Bitcoin's introduction, several other Blockchain platforms, extending the technology's applications, have emerged. In particular, Ethereum [11] can be seen as an extension of the Bitcoin Blockchain to support a wider range of applications. In order to establish trusted contracts without the need for a third party, Vitalik Buterin implemented the concept of Smart Contracts [12] into the Blockchain technology. What started out to be a secure payment consensus protocol, ended up evolving to the world's biggest decentralized distributed computer [13], with each node running its own instance of the Ethereum Virtual Machine (EVM) [14].

Due to the financial nature of the Smart Contracts running in the EVM, there are a lot of motivated hackers trying to find ways to exploit the system. Since once a contract is deployed it is immutable, not even its creator is able to patch any security vulnerability. In a study performed on nearly one million Ethereum contracts using MAIAN[1], 34,200 contracts were flagged as vulnerable [15]. In another study performed on 19,366 existing Ethereum contracts using Oyente[2], 8,833 were flagged as vulnerable, including the TheDAO[3] vulnerability which led to a 60 million US dollars loss in June 2016 [16].

In order to detect bugs and security flaws in their Smart Contracts' code [17, 18], developers use

[1]https://github.com/ivicanikolicsg/MAIAN
[2]https://github.com/enzymefinance/oyente
[3]TheDAO smart contract: https://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code

static analysis tools to analyse their code. These are ultimately necessary for software developers to find simple errors, serious vulnerabilities, performance issues, libraries' misuses[4], and even to standout design flaws[5].

According to a set of criteria [2], 10 static analysis tools were selected to be a part of SmartBugs [21]. SmartBugs was created with the purpose of being an extensible execution framework that simplifies the execution of analysis tools on smart contracts. It currently supports 10 Solidity static analysis tools and two smart contracts datasets.

Many static analysis tools have been created in academia but only a few are actively used by developers [22]. Developers mostly use the tools that have their results more easily accessible to them and are built to be integrated with the programming environment (i.e. Integrated Development Environments (IDEs)), unlike SmartBugs and any other Solidity static analysis tool. However, this integration is not trivial as different environments consume different formats for the static analysis' results, expect different plugins, and so forth.

1.2 Objectives and Contribution

In this thesis, we intend to overview smart contracts and their secure implementation utilizing static analysis tools to assess the code's correctness. Our major goal is to bring static analyzers, that focus on Solidity smart contracts, a step closer to being adopted by smart contract developers. With this, we hope to achieve a more effective use of these tools leading to less vulnerable programs being deployed.

To accomplish these goals, we will first provide an overview of blockchain technology before focusing on Ethereum smart contracts. We describe each of the top ten vulnerabilities reported in Solidity smart contracts.

Later, we present SmartBugs, a unique, extendable, and simple-to-use execution framework for smart contracts that facilitates the study and execution of static analysis tools. We implement our researched improvements into SmartBugs, mostly due to being a complex framework that aggregates a comprehensive different set of analysis tools.

For our improvements we start by breaking down a renowned standard for static analysis' results, the Static Analysis Results Interchange Format (SARIF). We present our overview on this standard and provide a minimalist way to implement it into any static analyzer's environment, for Solidity smart

[4]In 28-12-2020, an incorrect library usage, managed hackers to exploit $9.4 million US dollars from the *COVER* contract. By misusing the keywords *memory* and *storage*, the developers allowed for a serious security vulnerability in the *updatePool* function to be exploited [19].

[5]In 25-09-2020, the BZX developers noticed an increase in their *iToken* supply. This was due to a design flaw in their *transfer* function which allowed any user to increase their balance artificially by simply transferring money to himself [20].

contracts or any other type of code. By standardizing the output of an analysis tool, its creator is allowing for it to be integrated with other tools, i.e. GitHub.

Furthermore we research a possible methodology for creating a service that provides static analysis results when queried with an analysis request. We implement the Static Analysis Server Protocol (SASP) service with the goal of delivering the entire set of results to the client in the same format.

To conclude, we present an example of a platform that would digest our service's response and display it in a more intuitive manner. In order to improve the detection and awareness of the dangerous vulnerabilities that prey in Ethereum smart contracts, we implement all of our research into this platform in an automated form.

In summary, our major contributions are:

- We briefly review the vulnerabilities found in smart contracts and a state-of-the-art tool, SmartBugs, that can be used to automatically identify these vulnerabilities.
- We present an overview on SARIF, a standard for static analysis tools' output. We implement it into the SmartBugs' environment[6] and provide a more accessible way to implement it in other tools.
- We develop SASP[7] and build it as a service for reporting the SmartBugs' results to the clients who requested the analysis.
- We overview and program two GitHub adapters[8] for requesting static analysis results from SASP and directly from SmartBugs.
- Lastly, we discuss our implementations from different perspectives in order to provide an evaluation.

1.3 Thesis Outline

The rest of the document is divided into four chapters:

Chapter 2 provides an overview of the basis of blockchain technology, the Ethereum network, smart contracts, their vulnerabilities and SmartBugs. After this, Chapter 2 also describes the SARIF standard format, followed by the SASP service structure and the GitHub framework's multiple features which we will be using for our implementation.

[6]SmartBugs SARIF converters on GitHub: https://github.com/smartbugs/smartbugs/tree/master/src/output_parser
[7]SASP repository on GitHub: https://github.com/dindonero/smartbugs-sasp
[8]SASP GitHub Adapter: https://github.com/dindonero/smartbugs-static-analysis/
NoSASP GitHub Adapter: https://github.com/dindonero/smartbugs-local-static-analysis

Chapter 3 introduces our implementation for improving SmartBugs and the smart contract developers' community.

Chapter 4 presents our evaluation for each stage of our implementation.

Finally, Chapter 5 addresses work conclusions and proposes future work.

2 Background

Contents

2.1 Blockchain 9
2.2 Ethereum 11
2.3 Vulnerabilities of Ethereum Smart Contracts 12
2.4 SmartBugs 14
2.5 SARIF 19
2.6 SASP 27
2.7 GitHub 29

In this chapter we start by presenting an overview of the Blockchain technology, followed by explaining what smart contracts are and why it is critical to ensure proper smart contract's safety prior to its release. Furthermore we introduce SmartBugs as a unifying framework for smart contracts' analysis tools. We also present an overview of SARIF and SASP and how we will leverage this concepts into making SmartBugs a more accessible tool in order to improve Smart Contracts' security. Lastly, we present GitHub as the community chosen essential platform for managing their collaborative software projects.

This chapter is divided in six sections: Blockchain, Ethereum, Vulnerabilities of Ethereum Smart Contracts, SmartBugs, SARIF, SASP and GitHub

2.1 Blockchain

The name Blockchain comes from the fact that it is a growing list of blocks, each of them containing the hash of the preceding one. This way, the links between the blocks are cryptographically connected, making them essentially unbreakable. The only way to switch an older block would involve changing every single following block. In the Bitcoin implementation, each block is made up of a list of transactions that, once created and added to the blockchain, become immutable, meaning they can't be modified or reversed, guaranteeing the transactions' integrity. A transaction consists of the payer and the payee public keys, the bitcoin amount to be transferred and the digital signing of the transaction by the payer so that a payee can check the signatures in order to ensure the ownership is correct.

There are two types of blockchains: *Permissioned* and *Permissioneless*. Permissioned blockchains (or private blockchains) can be seen as an extra layer of security for a system, as they maintain an access control layer that allows only certain identifiable members to conduct particular tasks. It can be regarded as a centralized blockchain. On the other hand, permissionless blockchains (or public blockchains) can be seen as peer-to-peer, open networks available for everyone to participate and interact with, as long as they're willing to meet the consensus agreement. The network consists of a peer-to-peer decentralized architecture that allows it to persist even if one of its nodes is offline or attacked, increasing the system's trustworthiness. No node in this network trusts the others, yet due to the network's cryptographic features, they will always agree with each other by maintaining a shared global ledger, overcoming the lack of trust [9, 10].

Thanks to Satoshi Nakamoto, the pseudonym behind the Bitcoin paper, for proposing the first permissionless blockchain in 2008, the Bitcoin white paper [1]. Satoshi not only published the white paper as well as also developed some of the earlier code versions for the bitcoin network before disappearing in 2011. Up to this day Nakamoto's real identity still remains a mystery.

As it's a permissioneless blockchain and any user is able to interact and participate in the network by simply publishing a signed transaction, it raises one issue. The payee isn't able to verify if the payer did not double-spend the coin. Double-spending is a potential vulnerability in a digital cash system where the same digital token can be spent multiple times. In order to prevent this from happening, in the Bitcoin white paper, Satoshi suggested using the Proof-of-Work consensus mechanism as a way for the payee to know that the payer had not signed any other transactions, without the need for a third-party authority. The only way to certify the absence of a transaction is to be aware of all transactions so (1) all transactions must be publicly disclosed and (2) it's also required a system that enables participants to agree on the order that transactions were received.

Proof-of-Work The Proof-of-Work (PoW) algorithm entails searching for a value that, when hashed with SHA-256, begins with a number of zero bits. The average amount of work required is exponential to the number of zero bits required, and may be validated and verified with a single hash.

In the Bitcoin white paper, the PoW is implemented by incrementing a nonce in the block until a value is found that provides the block's hash the requested zero bits. The longest chain, with the most proof-of-work effort placed in it, represents the majority choice. So once the CPU effort to satisfy the proof-of-work has been expended, the block cannot be changed without redoing the work. And because later blocks are chained after it, changing a block would necessarily require redoing all of the blocks after it. We can see a visual representation provided in the Bitcoin white paper in 2.1.

Its difficulty is calculated based on a moving average targeting the average of blocks per hour to account for rising hardware speed and fluctuating interest in operating nodes over time. If the blocks are being generated too quickly, it becomes more difficult.

This work is done by a type of node called a *miner*. Once a miner has computed the right answer, the block is broadcast to the network for verification. The hash is checked by the other nodes in the network, and if it is valid, they accept the miner's block and the miner gets the reward.

There are also other protocols for consensus such as Proof-of-Stake. Juxtaposed to the PoW, Proof-of-Stake (PoS) validators (miners) can validate block transactions based on the number of coins they own. To verify a block of transactions, the validator stakes some of his currency, and the larger the amount staked, the more validation power the validator has.

PoS cryptographic calculations are substantially easier to solve than PoW ones, and so do not require as much computer power. Proof-of-stake systems are more cost-effective and consequently more environmentally friendly than Proof-of-Work systems in terms of energy consumption, as they do not require the significant computer power that Proof-of-Work does. In terms of security, proof of stake can help lower the likelihood of malicious attacks. Attacking in PoS is costly because if a validator behaves

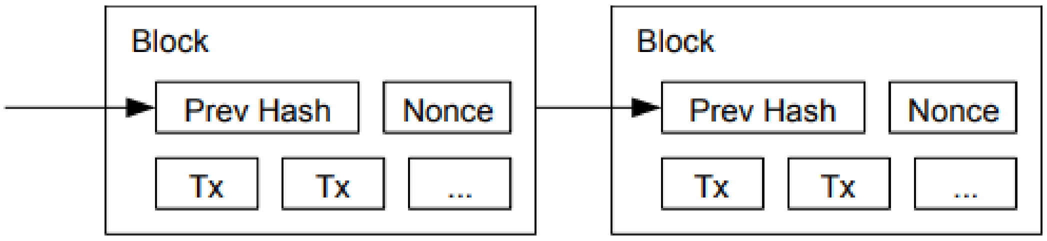


Figure 2.1: PoW Chain of Blocks (Adapted from: [1])

maliciously, their validation authority will be withdrawn and their staked amount will be revoked. Also the amount staked is directly proportionate to the percentage of network control a member can achieve. On the other hand in PoW, the more a miner invests in equipment, the more processing power they will have, making individual mining against massive mining pools less profitable and more challenging. In Bitcoin, pools mine a disproportionately large number of blocks compared to individual miners. A 2019 study showed that the 3 biggest mining pools control over 50% of the hash rate and that in each of them, 18 miners receive more than 50% of the pool rewards payout [23].

2.2 Ethereum

Since Bitcoin's launch, other Blockchain platforms have developed, expanding the technology's possibilities. Specifically Ethereum [11] may be perceived as a more robust version of the Bitcoin Blockchain which enables a broader range of applications. It's a project that aims to create a generalized technology that can be used to build all transaction-based state machine concepts. It also aims to provide end-developers with a tightly integrated end-to-end system for developing software using a previously unexplored computer paradigm: a trustworthy object messaging service framework [24].

Vitalik Buterin introduced Ethereum in 2014 as the first blockchain platform to incorporate a Turing-complete language. It has its own currency, called Ether (ETH), with smart contracts as the platform's key component. Just like Bitcoin, Ethereum is a permissioneless blockchain where any network participant is allowed interaction with the network, such as transferring Ether or interacting with a smart contract.

2.2.1 Smart Contracts

The idea of Smart Contracts is not new. In fact, it's been here for 25 years, with Nick Szabo defining the concept in 1996. A smart contract, according to Szabo, is *"a set of promises, specified in digital form,* *including protocols within which the parties perform on these promises"* [25].

With the emergence of blockchain technology, Szabo's concept of smart contracts became a possibility. And nowadays smart contracts are pieces of code that are executed on a Blockchain to formalize agreements in order to obtain trust from the multiple distrusting parties.

When consensus on the outcome of the execution of a smart contract is achieved by the network of nodes, the contract's state gets updated and stored on the blockchain. Smart contract processing sometimes demands intensive computing operations, and because these tasks are carried out by the networks' nodes, each computational operation has a cost, which in Ethereum is referred to as *gas*. These execution fees are the costs incurred by the user in having code executed by the miners. Users can engage with contracts and trade value or data by posting signed transactions to the network.

The reasons that make smart contracts such a powerful premise, the lack of need for confidence between parties involved and their immutability, can also be viewed as a flaw that must be carefully managed. Any flaw or error in a smart contract's programming can be exploited. A smart contract with vulnerabilities cannot be patched because the code is nearly impossible to correct. Code is law in the blockchain world. Before the contract is deployed, rigorous code verification is required to ensure that it is as secure.

Solidity [26] is the world's first Turing complete programming language [27] to operate in a Blockchain platform. With a structural design similar to the JavaScript language, Solidity compiles to EVM byte code [14] so it can be executed by the multiple nodes running the EVM.

Accounts on Ethereum can be externally owned (EOAs) or contract-based. Like bitcoin wallet addresses, EOAs are controlled by private keys that can be used to make transactions and sign data. The account is fully controlled by the owner of the private key, while on the contract-based account it's controlled by the smart contract code. They can receive funds and conduct transactions in the same way that EOAs may, but there is usually no single private key used to conduct transactions. Instead, smart contract code defines the logic that governs how transactions are carried out [28].

2.3 Vulnerabilities of Ethereum Smart Contracts

There are numerous types of vulnerabilities in Ethereum smart contracts, and they can manifest themselves in various ways. Throughout SmartBugs' research, they chose the most comprehensive and prevalent way of describing the vulnerabilities observed in smart contracts, based on the Decentralized Application Security Project (DASP) Top 10 taxonomy [29]. All of SmartBugs' tools reported vulnerabilities were mapped into one of those categories.

We will overview each of the DASP 10 categories:

1. **Reentrancy** - It is regarded as the most serious smart contract vulnerability discovered to date, leading to one of the largest attacks on the Ethereum network, the DAO [30]. A reentrancy exploit is performed by altering the program flow in a way that the smart contract creator did not intend. By making reentrant calls that take use of a intermediary state, a malicious attacker can withdraw several times and deplete the balance of a contract.

2. **Access control** - This vulnerability is ubiquitous in all programs, not just smart contracts. An attacker can modify contracts if the function modifiers aren't correctly used to prevent access. On a smart contract, all important functions must be properly safeguarded. The largest attack on smart contracts, the Poly Network attack [31], can be categorized as an access control vulnerability. In August 10 of 2021, the attacker was able to steal over 600 million dollars. Even though the actual attack occurred on the Poly network, it also affected other blockchains (i.e. Ethereum, Binance) because the Poly network is a cross chain network [32].

3. **Arithmetic Issues** - This type of vulnerability, often known as integer overflow and underflow, is not new and is ubiquitous to many programming languages. As a precaution, all arithmetic data should be validated, and therefore trusted mathematical libraries, such as the SafeMath module, should be used.

4. **Unchecked Low Level Calls** - Low level calls should be avoided wherever feasible since if they fail, they will return a boolean value instead of a complete rollback of the present execution, while the code will continue running. When return values are not handled properly, undesirable behavior can occur.

5. **Denial of Service (DoS)** - This particular attack involves flooding the network with time-consuming computations, causing the contract to become unusable for a short amount of time or indefinitely.

6. **Bad Randomness** - The blockchain provides no cryptographically safe library of randomization, everything in a contract is publicly available, and the nodes will always outperform the blockchain. Although there are a number of methods and variables which can create difficult values to predict at first glance, they are usually not as clear as they appear, and might be more exposed than they would seem to be influenced by miners.

7. **Front Running** - When multiple dependent transactions that trigger the same contract are combined into a single block, front running issues may occur. Since transactions are only deemed valid when they have been validated by a miner, it's him who determines the sequence in which transactions are executed. If the transactions were not performed in the correct order, a malicious participant may launch an attack. One possible method would be by overpaying the miner so that he processes the attacker's transaction first.

8. **Time manipulation** - The miner can change the block's timestamp, therefore all intended uses of the timestamp in the smart contract's code should be careful reviewed.

9. **Short Addresses** - Because the EVM accepts poorly defined arguments, short address attacks are conceivable. It can be used to make poorly programmed clients encode arguments improperly before adding them in transactions by utilizing specially designed addresses.

10. **Unknown Unknowns** - Smart contracts, the platforms and its programming languages are still in their infancy, and their implementation and use are very new. New types of vulnerabilities will continue to be discovered, and developers must stay aware and up to date. All vulnerabilities that don't fit any of the other categories are part of the Unknown Unknowns.

2.4 SmartBugs

The academic community has been working on developing automated analysis tools to find and eliminate vulnerabilities in smart contracts [2, 16, 33–37]. However, it is difficult to compare and replicate that research since, while some of the tools are open source, the datasets used are not. Furthermore, most static analysers for smart contracts must be installed, and some even demand you to install dependencies on which they rely. Others aren't compatible with all operating systems. Performing smart contract analysis with many tools, running and installing each one separately, can be a tedious procedure that wastes time. As a solution, a tool was developed in 2019 to make static analysers easier to use and to give a simple interface via which the user could analyze various contracts with multiple tools without having to install any of them.

SmartBugs is an extensible and simple-to-use execution framework for smart contracts developed in Solidity that facilitates the research and execution of different automated analysis tools [2, 21].

Its code[1] is written in Python 3 and the tools are run using Docker images. These images can be found on the Docker Hub[2] or locally. The decision to adopt docker images was made to simplify the inclusion of tools, to allow for reproducible execution, and to maintain the same execution environment for all tools, allowing the user to run SmartBugs in any environment that has Python3 and Docker installed. This way, any developer can add their new tool and effortlessly compare it against existing ones.

SmartBugs is made up of the following core components (We can see a visual representation in Figure 2.2):

1. The Command-Line Interface (CLI),

[1]SmartBugs (including $SB^{CURATED}$): https://github.com/smartbugs/smartbugs
[2]Docker Hub: https://hub.docker.com/

2. The Web-based User Interface (WUI),
3. The tools' plugins,
4. The Docker images that are hosted on Docker Hub,
5. The datasets of smart contracts,
6. The SmartBugs' program. The component that connects all of the SmartBugs components in order to run the analysis tools.

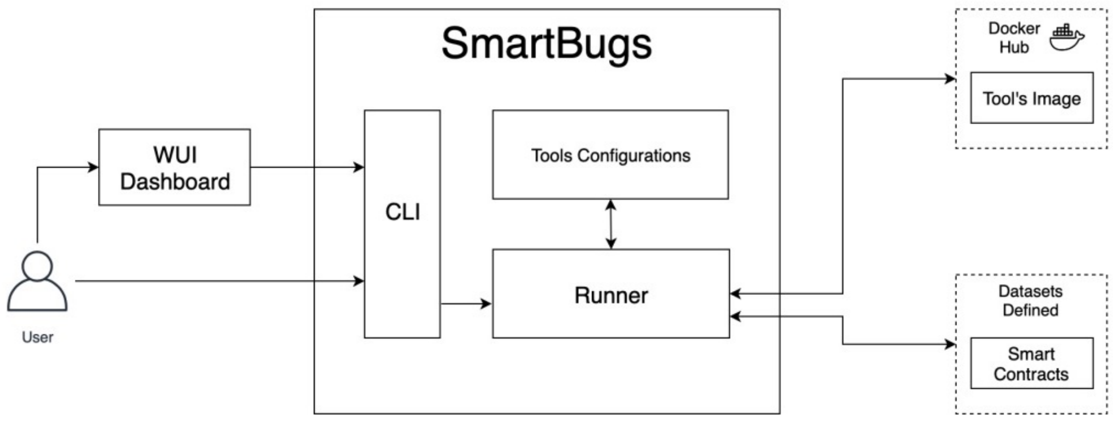
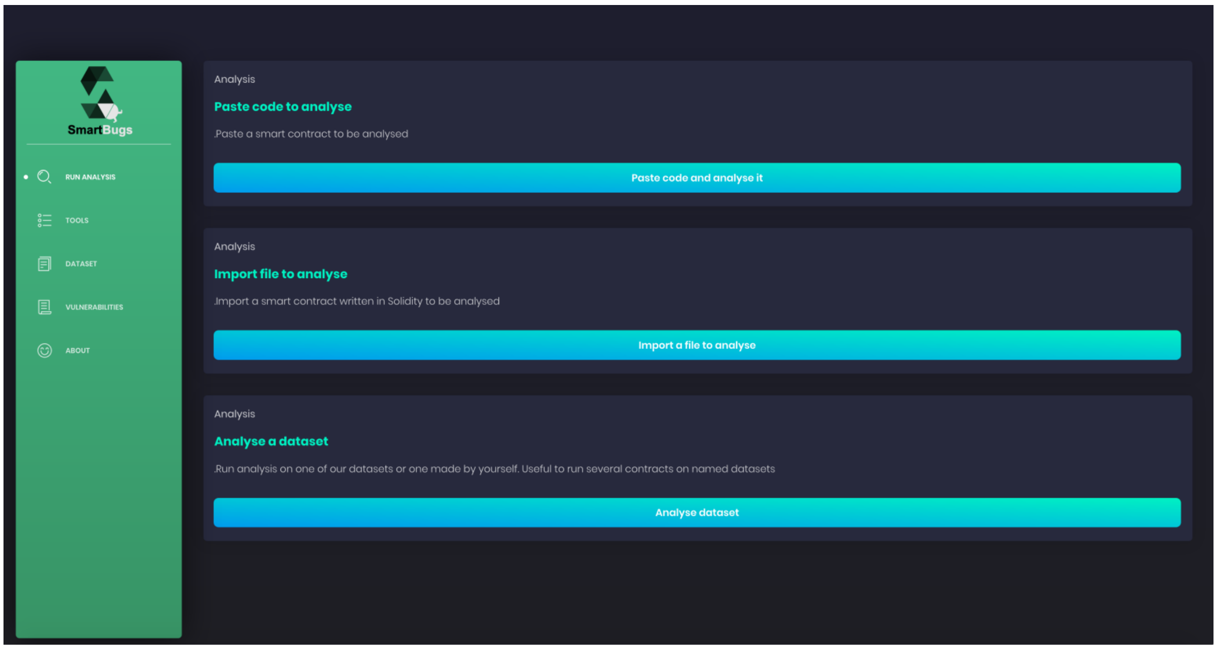


Figure 2.2: SmartBugs Architecture (Adapted from: [2])

2.4.1 Web-based User Interface

The WUI Dashboard is implemented on top of SmartBugs. This dashboard gives the user easier and more intuitive access to the tools' list, named datasets, and the vulnerabilities found by each tool, which are all mapped to a DASP 10 category. However it's not dynamically updated, which means that if new tools or datasets are introduced to SmartBugs, they will not appear in the WUI automatically (unlike CLI). The dashboard supports ten tools, the $SB^{CURATED}$ dataset, and gives a complete mapping of all vulnerabilities found by the ten tools. A preview of the SmartBugs' WUI Dashboard is shown under Figure 2.3.



Analysis
Paste code to analyse
.Paste a smart contract to be analysed
Paste code and analyse it

Analysis
Import file to analyse
.Import a smart contract written in Solidity to be analysed
Import a file to analyse

Analysis
Analyse a dataset
.Run analysis on one of our datasets or one made by yourself. Useful to run several contracts on named datasets
Analyse dataset

Figure 2.3: SmartBugs' WUI Dashboard

2.4.2 Tools

SmartBugs' authors gathered 35 possible suitable tools from a survey [38] and through research. From those 35 state-of-the-art analysis tools, only the following 10 were picked to be integrated into Smart-Bugs' environment:

1. **HoneyBadger** [39] is a tool that detects honeypots in smart contracts using symbolic execution and a set of heuristics. Honeypots are smart contracts that appear to have an evident vulnerability in their design, allowing an arbitrary user to steal Ether from the contract. All the Ether spent by the user trying exploiting the contract is kept by the contract as it's not really vulnerable.

2. **Maian** [40] analyses contracts for three main errors. Suicidal, Prodigal and Greedy contracts. Suicidal contracts enable an attacker to destroy the entire contract and return the funds to its owner. A prodigal contract possesses a flaw that enables any person to hijack the contract and send the funds to themselves. Greedy contracts allow an attacker to lock Ether forever, like in Ethereum Parity Wallet hack [41].

3. **Manticore** [42] is a tool that implements constraint solving, a technique based on dynamic symbolic execution to systematically explore a smart program's state space.
4. **Mythril** [43] analyzes EVM byte-code and employs concolic analysis, a hybrid analytic technique that runs symbolic execution along a specific execution route, to detect several types of security vulnerabilities.
5. **Osiris** [44] combines symbolic execution and taint analysis to reliably identify integer errors in Ethereum smart contracts.
6. **Oyente** [16] is a static analysis tool that has been forked by multiple other projects, such as HoneyBadger and Osiris. It performs symbolic execution on EVM byte-code to detect vulnerabilities.
7. **Securify** [34] analyzes EVM byte-code using the Souffle datalog solver in order to derive meaningful and accurate sematic information about the smart contract.
8. **Slither** [45] is a static analysis framework that translates Solidity into an intermediate representation called SlithIR and extracts and refines information using well-known program analysis techniques like dataflow and taint tracking.
9. **SmartCheck** [33] is a static analysis tool that detects security flaws and poor coding practices. It analyzes Solidity source code for lexical and syntactical flaws.
10. **Solhint** [46] is a linting solution tool that detects syntax-related security code vulnerabilities.

2.4.3 Datasets

As the smart contracts datasets are not publicly available, if a developer wants to test its new tool and compare it to existing work, it would be necessary to contact the authors of alternative tools and hope that they would give access to their datasets. To address this, SmartBugs supplies two smart contracts datasets.

The $SB^{CURATED}$, a dataset of 143 manually annotated Solidity smart contracts which can be used to test the precision of analysis tools, in order to be able to execute and compare automated analysis methods, thus laying the groundwork for fair comparisons. It was built by combining smart contracts from three separate places: GitHub repositories, contract-analysis blog postings, and the Ethereum network. The vast majority of contracts came from GitHub repositories and the Ethereum network.

This dataset comprises 143 vulnerable smart contracts with 191 manually labeled vulnerabilities, categorized into the DASP 10 taxonomy, which we reviewed under Section 2.3. For each DASP 10 vulnerability category, the Table 2.1 provides the number of contracts, vulnerabilities and lines of code of that category.

Category	**Contracts**	**Vulns**	**LoC**
Access Control	17	19	899
Arithmetic	14	24	304
Bad Randomness	8	31	1,079
Denial of service	6	7	177
Front running	4	7	137
Reentrancy	31	32	2,164
Short addresses	1	1	18
Time manipulation	5	7	100
Unchecked low level calls	53	60	4055
Other	3	3	115
Total	143	191	9,048

Table 2.1: Categories of vulnerabilities available in the $SB^{CURATED}$ (Source: [2])

The SB^{WILD} has a total of 47,518 contracts and contains all unique Solidity smart contracts in the Ethereum Blockchain that have their source code available in Etherscan[3]. It was gathered by the co-authors of [47] and it contains all the different contracts from Ethereum, because Etherscan allows anyone to download any contract's source code as long as it's been made available and you know its address.

2.4.4 Analysis

The biggest experimental execution on smart contract static analysis was performed in [21], both in terms of tool quantity and execution time. Seven tools were used to do 333,109 analyses, which took a total of 330 days and 15 hours to complete, nearly a full year of continuous analysis. All of the logs and raw analysis findings are accessible in the SmartBugs[4] repository on GitHub.

It was also performed an experimental analysis using the same tools on the $SB^{CURATED}$ dataset with the purpose of calculating the seven tools' ability to identify vulnerabilities in the 69 contracts. All of the logs and raw analysis findings are accessible in the SmartBugs[5] repository on GitHub.

The accuracy result produced by the tools in this analysis is shown in Table 2.2. It shows the amount of vulnerabilities that have been detected based on the following criteria: A vulnerability is deemed as

[3] https://etherscan.io/
[4] SB^{WILD} analysis results: https://github.com/smartbugs/smartbugs-wild
[5] SB^{WILD} analysis results: https://github.com/smartbugs/smartbugs-results

correctly detected when a tool identifies a vulnerability of a specific category at a specific line, and it matches the same vulnerability that has been annotated in the dataset [48].

Category	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0/19 0%	0/19 0%	4/19 21%	4/19 21%	0/19 0%	0/19 0%	0/19 0%	4/19 21%	2/19 11%	5/19 26%
Arithmetic	0/22 0%	0/22 0%	4/22 18%	15/22 68%	11/22 50%	12/22 55%	0/22 0%	0/22 0%	1/22 5%	19/22 86%
Denial Service	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/7 0%	0/ 7 0%
Front Running	0/7 0%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	2/ 7 29%
Reentrancy	0/8 0%	0/8 0%	2/8 25%	5/8 62%	5/8 62%	5/8 62%	5/8 62%	7/8 88%	5/8 62%	7/ 8 88%
Time Manipulation	0/5 0%	0/5 0%	1/5 20%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	2/5 40%	1/5 20%	3/ 5 60%
Unchecked Low Calls	0/12 0%	0/12 0%	2/12 17%	5/12 42%	0/12 0%	0/12 0%	3/12 25%	4/12 33%	4/12 33%	9/12 75%
Other	2/3 67%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	0/3 0%	3/3 100%	0/3 0%	3/ 3 100%
Total	2/115 2%	0/115 0%	13/115 11%	31/115 27%	16/115 14%	17/115 15%	10/115 9%	20/115 17%	13/115 11%	48/115 42%

Table 2.2: Accuracy of the 7 tools based on the vulnerabilities detected on the $SB^{CURATED}$ analysis' result (Source: [2])

Among the seven tools, Mythril has the best accuracy. While the average of all tools is 12 percent, Mythril correctly identifies 27 percent of all vulnerabilities. The tools that detect the most different types of categories are Mythril, Slither, and SmartCheck as they detect 5 different categories. The authors also combined the tools together to calculate the best performance as presented in Table 2.3. As a result, they recommend combining Mythril and Slither, as it detects 37 percent of all vulnerabilities and provides a decent ratio of performance and execution cost. The second best pair, Mythrill and Oyente, identifies 29 percent of all vulnerabilities.

	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck
Honeybadger		2/115 2%	15/115 13%	33/115 29%	18/115 16%	19/115 17%	12/115 10%	20/115 17%	15/115 13%
Maian			13/115 11%	31/115 27%	16/115 14%	17/115 15%	10/115 9%	20/115 17%	13/115 11%
Manticore				32/115 28%	26/115 23%	26/115 23%	19/115 17%	27/115 23%	20/115 17%
Mythril					33/115 29%	33/115 29%	31/115 27%	42/115 37%	33/115 29%
Osiris						22/115 19%	21/115 18%	31/115 27%	23/115 20%
Oyente							22/115 19%	32/115 28%	25/115 22%
Securify								25/115 22%	16/115 14%
Slither									25/115 22%
Smartcheck									

Table 2.3: Combined accuracy of the 7 tools based on the vulnerabilities detected on the $SB^{CURATED}$ analysis' result (Source: [2])

2.5 SARIF

2.5.1 Standard OASIS

Organization for the Advancement of Structured Information Standards (OASIS)[6] is a nonprofit organization that develops open standards for multiple areas such as Blockchain, Cloud computing, emergency management, and so on. It was founded as SGML Open to promote their standard for defining generalized markup languages for documents, SGML. Later with the community adoption of XML as its

[6] https://www.oasis-open.org/

standard, SGML Open changed their name to OASIS Open and started working on standards for a variety of other topics. OASIS is composed of members from various companies such as Microsoft, Oracle, IBM and individuals and employees from other companies.

After developing several accepted stardards like ebXML[7], for the global use of electronic business information, OData[8], for the creation and consumption of queryable and interoperable REST API, UDDI[9] and others, they developed SARIF and SASP [49].

2.5.2 SARIF - A comprehensive analysis standard

Common static analysis tools usually generate outputs in their own distinct format. As a result, software developers are faced with the task of parsing and aggregating various format outputs so that they can comprehend and acknowledge its information. In order to unify the output format of distinct static analysers, a standard file format for exchanging results was proposed. SARIF originated at Microsoft, and is now a standard being developed under OASIS. The technical committee has members from several static analysis tool vendors and large-scale users [49].

SARIF is JavaScript Object Notation (JSON) based format and aggregates all possible information about an analysis. From its results, to its metadata such as schema, version and URI, and even to the execution path, SARIF's standard concentrates all this information in a single file. It has been increasing its popularity as recent developed tools are exporting its outputs under this format and older tools are converting to its standard (i.e. CogniCrypt[10], Clang Static Analyzer[11] and Pylint[12]) [50, 51].

In this document we provide an overview of SARIF 2.1.0 [52].

2.5.2.A Root Structure

The SARIF's standard format [52] is composed of following three main root keys, as shown in Listing 2.1:

1. ***version*** - This parameter identifies the SARIF's format version (currently v2.1.0).
2. ***$schema*** - Specifies the URI of the JSON Schema for the given version.
3. ***runs*** - The heart of the analysis, an array that contains all the information related to the analysis where each object corresponds to a different static analysis tool.

[7]ebXML: `https://www.ebxml.org/`
[8]OData: `https://www.odata.org/`
[9]UDDI: `https://uddi.xml.org/`
[10]CogniCrypt: `https://www.eclipse.org/cognicrypt/`
[11]CSA: `https://clang-analyzer.llvm.org/`
[12]Pylint: `https://www.pylint.org/`

Listing 2.1: SARIF Root Object

```
{
    "version": "2.1.0",
    "$schema": "http://json.schemastore.org/sarif-2.1.0-rtm.4",
    "runs": [ 2.2 ]
}
```

The parameter ***runs***' values, which are shown under Listing 2.2, are divided into two categories, the analysis results, composed of the keys *artifact*, *invocations*, *results*, and *logicalLocations*, and the analysis metadata, composed of the *tool* key [53].

Listing 2.2: SARIF *runs* subkeys

```
"runs": [
    {
        "artifacts": [ 2.3 ],
        "invocations": { 2.4 },
        "logicalLocations": [ 2.5 ],
        "results": [ 2.6 ],
        "tool": { 2.7 }
    }
]
```

2.5.2.B Analysis Results

The analysis results details the information about the analysis itself, what was analyzed, how it was analyzed, when it was analyzed and the analysis' results. These are detailed below:

Note: All snippets shown below are created based on Oyente [16], one of SmartBugs' static analysis tool, analyzing the contract SimpleDao [30]. Due to the lack of output information produced by the tools, these examples correspond to a more complete output compared to ours. These examples serve as a global example to incorporate the maximum of information into SARIF from a future added tool. The actual SARIF output for the analysis of SimpleDAO produced by Oyente can be found in Listing 3.1.

- ***artifacts*** - A list containing all files examined by the tool (even if results were not detected). The

URI string must not start with a backslash and must be relative to the repositories root. An example for the artifact key is shown in Listing 2.3.

Listing 2.3: SARIF *artifacts* subkeys

```
    "artifacts": [
        {
            "location": {
                "uri": "dataset/reentrancy/simple_dao.sol"
            },
            "sourceLanguage": "Solidity",
            "hashes": {
                "sha-256": "b13ce2678a8807ba0765ab94a0ecd394f869bc81"
            }
        },
    ]
```

- ***invocations*** - Describes the invocation information of the analysis which mainly includes the start and end time - represented under the ISO 8601:2004 [54] - the environment variables used, the tool execution and configuration notifications and the command used to run the analysis. SARIF uses the toolConfigurationNotification when describing conditions relevant to the tool and toolExecutionNotification when describing the runtime environment. An invocation representation example is shown in Listing 2.4.

Listing 2.4: SARIF *invocations* subkeys

```
    "invocations": {
        "commandLine": "python oyente.py -s dataset/reentrancy/
            simple_dao.sol",
        "responseFiles": [
            {
                "uri": "dataset/reentrancy/simple_dao.sol",
                "uriBaseId": "SRCROOT",
            },
        ],
        "startTime": "2020-11-16T15:18:30Z",
        "endTime": "2020-11-16T15:20:15Z",
        "fileName": "simple_dao.sol",
        "workingDirectory": {
            "uri": "file:///home/smartbugs/"
        },
        "environmentVariables": {
            "PATH": "..",
            "HOME": "..",
        },
        "toolConfigurationNotifications": [
            {
                "level": "note",
                "message": {
                    "text": "Smart Contract Compiled Successfully"
                }
            }
        ],
        "toolExecutionNotifications": [
            {
                "level": "note",
                "message": {
                    "text": "Done [1/1]: dataset/reentrancy/
                        simple_dao.sol \n Analysis completed."
                }
            },
        ]
    }
```

- ***logicalLocations*** - This key contains information yielded by the static analysis tool about logical location (e.g. contract name and functions). An example is shown in Listing 2.5.

Listing 2.5: SARIF *logicalLocation* subkeys

```
    "logicalLocations": [
        {
            "kind": "contract",
            "name": "SimpleDAO"
        },{
            "name": "withdraw",
            "fullyQualifiedName": "SimpleDao.withdraw",
            "kind": "function"
        }
    ]
```

- ***results*** - This array contains the results yielded by the static analysis tool. Each object contains a defect reported by the analysis. The *results* key can be the most complex of the *runs* object and as such we will define it next. A visual example for the results object is shown in Listing 2.6
  - ***ruleId*** - The distinctive identifier for the rule evaluated to produce the result. This key must be unique for each of the vulnerabilities reported.
  - ***message*** - Describes the defect detected by the static analysis tool.
  - ***level*** - Expresses the severity of the results. It's composed of one of the following values:
    1. *warning* - Meaning a problem was found.
    2. *error* - Meaning a serious problem was found.
    3. *note* - Meaning a minor problem was found or a possibility to improve the code.
    4. *none* - Meaning no problem was found.
  - ***locations*** - An array where each object describes an exact location for the defect found. It can be expressed both as a physicalLocation (e.g. file name, line and column number) and/or as a logicalLocation (as we have seen previously). We recommend always using the physicalLocation as it's the one that most consumers digest.

Listing 2.6: SARIF *results* subkeys

```
    "results": [
        {
            "ruleId": "Reentrancy_6"
            "kind": "fail",
            "message": {
                "text": "Re-Entrancy Vulnerability."
            },
            "level": "warning",
            "locations":
            [
                {
                    "physicalLocation": {
                        "artifactLocation": {
                            "uri": "dataset/reentrancy/simple_dao.sol
                               ",
                        },
                        "region": {
                            "startLine": 19,
                            "endLine": 19,
                            "startColumn": 18,
                            "endColumn": 23,
                            "snippet": {
                                "text": "bool res = msg.sender.call.
                                   value(amount)();"
                            }
                        }
                    },
                    "logicalLocation": {
                        "fullyQualifiedName": "SimpleDAO.withdraw",
                        "index": 1
                    }
                }
            ]
        }
    ]
```

2.5.2.C Analysis Metadata

The analysis metadata specifies all information regarding the tool that run the analysis and produced its results. The *tool* key is composed of the name, version, description and an array containing the detailed rules for the warnings found by the analysis. It may also contain an array for the detailed notifications yielded in the *results* and an URI for the tool to help the developer responsible for addressing the result. Each rule should also contain a name identifier for uniformity of experience across all tools that produce SARIF. The friendly name should be a single Pascal formatted identifier[13]. An example for the object *tool* is shown in Listing 2.7.

Listing 2.7: SARIF *tool* object

```
    "tool": {
        "name": "Oyente",
        "version": "0.4.25",
        "informationUri": "https://oyente.tech/",
        "fullDescription": {
            "text": "Oyente runs on symbolic execution, determines
                which inputs cause which program branches to execute,
                to find potential security vulnerabilities. Oyente
                works directly with EVM bytecode without access high
                level representation and does not provide soundness
                nor completeness."
        },
        "rules": [
            {
                "id": "Reentrancy_6",
                "name": "ReentrancyVulnerability",
                "shortDescription": {
                    "text": "Re-Entrancy Vulnerability."
                }
                "fullDescription": {
                    "text": "A possible Callstack Depth Attack was
                        found on the code thus producing a Re-Entrancy
                         Vulnerability"
```

[13] What is a Pascal formatted string?

```
                        }
                    }
                ]
            }
```

For a more simple and step-by-step tutorial on how to replicate SARIF and apply it to a tool, please refer to the docs in [55].

2.6 SASP

SASP [49] is a standardized communication protocol for communicating the static analysis tools' results to their consumers. Furthermore it is also designed for batch execution of analysis tools to actively communicate the results between each other. Basically, SASP acts as a service where clients (i.e. IDEs) can request results from static analysis tools to be integrated in the code. For the protocol to respond to this query quickly, it needs a common output standard to aggregate all analysis results efficiently. It achieves this by strongly leveraging SARIF. Currently, static analysis tools need an ad-hoc point-to-point communication between them and the developers' tools. A visual example for the currently needed connections is shown in Figure 2.4.

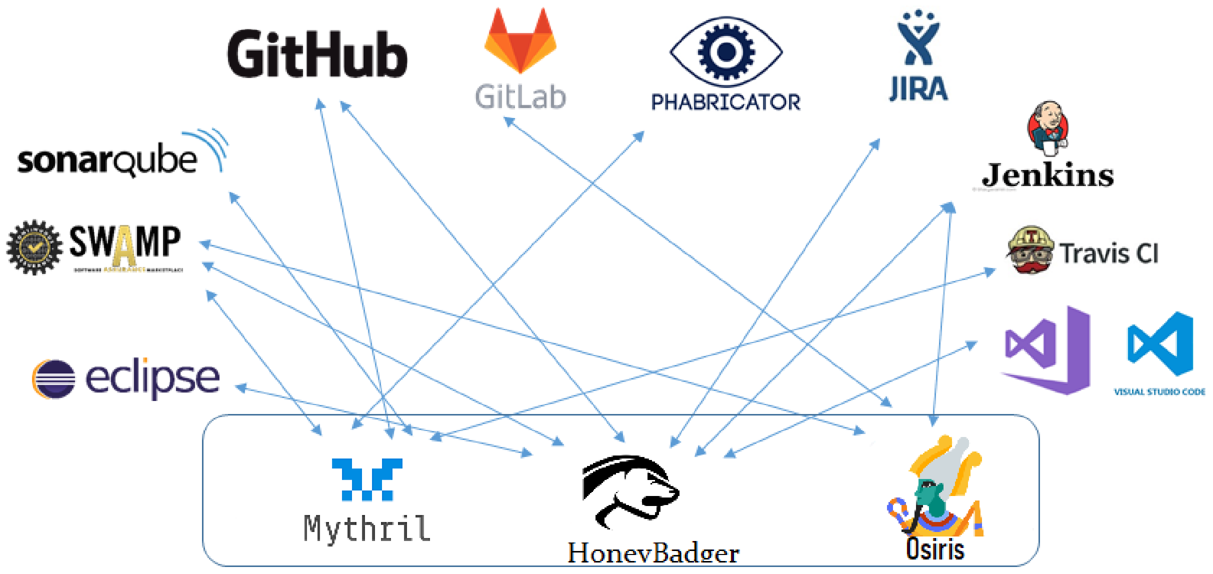


Figure 2.4: Point-to-point connections between static analysis tools and developer tools (Adapted from: [3]).

Opposed to this example, we introduce SASP's vision, where both static analysis tools and developer's platforms would communicate through SASP in order to aggregate all analysis tools outputs and produce a standardized result ready to be consumed by the developer's tool that made the request. An example for this vision is shown in Figure 2.5.

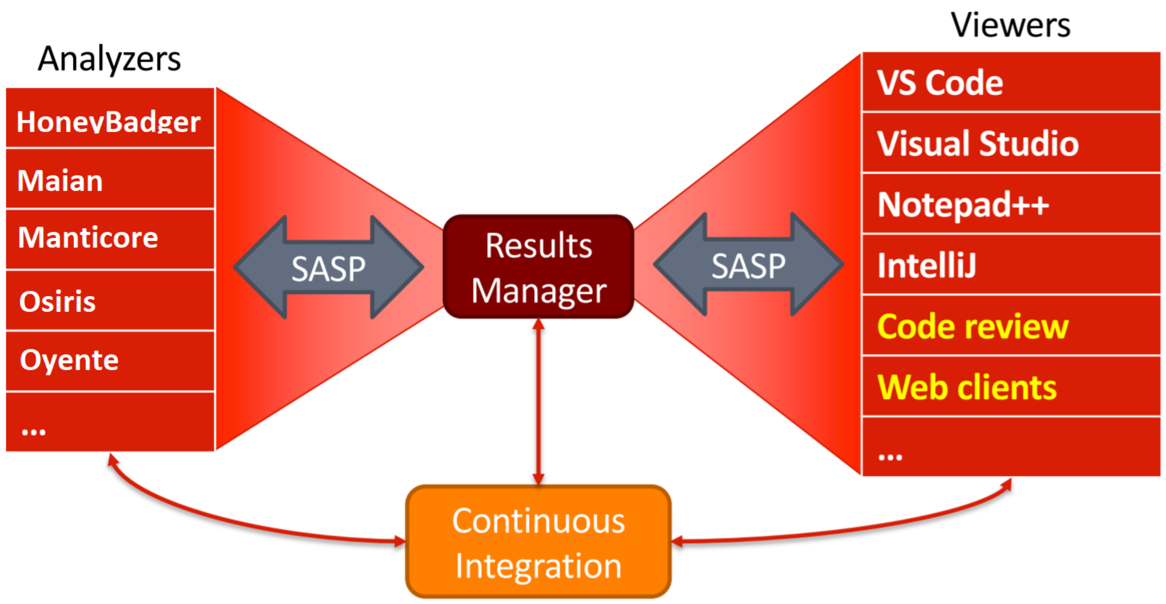


Figure 2.5: SASP Vision: Using SASP for all communications between static analysis tools and developer tools (Adapted from: [3]).

This way, SASP can be integrated with:

- IDEs and Editors - To overlap static analysis results on the code.
- Code Review Systems - Like GitHub's pull request system.
- Results Analytics - To create a report or dashboard of multiple static analysis tools.
- Bug Tracking - To see if a defect is detected in an analysis.
- Continuous Integration - To indicate a status of a build, for example, if there is a serious security vulnerability it would report the build has "failed".

Generally, SASP promises a more explicitly fine-tuned integration compared to other existing server protocols (i.e. Language Server Protocol[14]) [51], which in turn means more work in the server. However SASP is still in the early stage of its development and is still missing a formal specification for the protocol itself.

[14]LSP: https://microsoft.github.io/language-server-protocol/

2.7 GitHub

GitHub[15], GitLab[16], Gitorius[17], BitBucket[18] are all web hosting applications for collaborative software development projects with a few differences between each other. More and more frequently developers program collectively in these online platforms. Either is because modern programs require large bodies of code, or because teams are distributed around the globe or simply due to being more practical to have all your code centralized in one place. We chose GitHub as the ecosystem to further integrate SmartBugs given that it is the only that aggregates all following attributes:

1. It's free of use where the free version has the maximum limits compared to each of the platforms (i.e. max users for a project, max amount of storage used).
2. Provides an intuitive version control system.
3. Is the platform with the most amount of users (over 56 million developers) and the most amount of projects (over 190 million repositories) making it the largest host of source code in the world.
4. Features a SARIF processing mechanism for vulnerabilities scanning. [56]
5. Is the only platform with GitHub Actions[19] which will greatly facilitate integrating a SASP service into GitHub's workflow.
6. Features an Online Martketplace where users can add actions to their repositories with a single click.

2.7.1 Workflow

First, we are going to start with an overview of GitHub's workflow.

Branch GitHub is a branch based workflow platform. This means that a project has a main branch that should be ready for deployment and that it also has secondary branches, created by the owner or any other developer, with the goal of improving the code. When a developer has a different idea or feature that he wants to add in a project, he starts by creating a new branch. This technique exists to help developers managing a project's workflow. When this branch is created, it's created a new environment where developers can try out their new ideas. As the changes they make won't affect the

[15]GitHub: `https://github.com/`
[16]GitLab: `https://about.gitlab.com/`
[17]Aquired by GitLab in 2015. Former: `https://gitorious.org/`
[18]BitBucket: `https://bitbucket.org/`
[19]GitHub Actions: `https://docs.github.com/en/actions`

main branch, developers are free to experiment and commit their code, always knowing that their code won't get merged before the project's collaborators review and accept their changes.

Commit After creating the new branch, a developer can add, edit or delete a file. When he adds these changes to his new branch, he is making a commit. Commits create a clear history of a developer's work so that his collaborators can easily understand what he has done. Each commit can be associated with an descriptive message in order to explain why the changes were made. Moreover, each commit is able to be rolled back in case an error or bug are later found.

Pull Request Pull Requests start a discussion about the commits made inside that branch. Everyone is able to see all the changes made in a fairly intuitive interface. A developer can start a pull request at anytime during development. If a developer is stuck or has a question, he can also start a pull request even without code changes. Ultimately these are started when a developer is ready for someone to review their work and he can even mention specifically a user or a team for feedback.

Discussion and Review After opening a pull request, the team reviewing the changes might have comments or questions. Perhaps the design is not up to the project's standards, the changes proposed may be failing some tests or everything is in order and good to go. Pull Requests are made to encourage this discussion. In between discussions, a developer is still able to push some code, specially after someone pointing out a bug or flaw. GitHub will show the new commits and feedback in an unified and intuitive Pull Request view. This is also where we recommend for a developer to run the SmartBugs' analysis.

Deploy GitHub allows deployment of the branch before merging to the main branch for final testing in production. After a Pull Request's changes passes the tests and is reviewed, a developer can deploy the changes to test the final product. If the changes cause issues, the developer can still roll back by deploying the main branch.

Merge Finally that all changes have been reviewed and verified, a developer can now merge its code into the main branch. After merging, Pull Requests keep a history of the changes made so anyone can search for a specific one, go back in time to that point of the repository and even revert all changes if they decide to take the project on another direction.

A visual representation of the GitHub's workflow is shown in Figure 2.6. The workflow's step in purple is where we recommend to integrate SmartBugs.

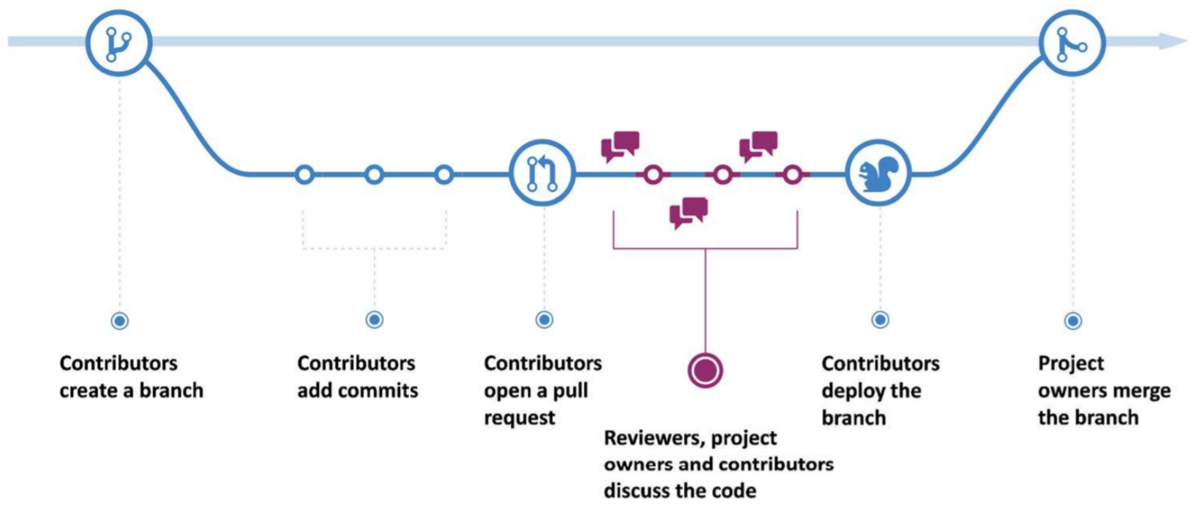

Contributors create a branch
Contributors add commits
Contributors open a pull request
Reviewers, project owners and contributors discuss the code
Contributors deploy the branch
Project owners merge the branch

Figure 2.6: GitHub's new branch workflow. SmartBugs automated analysis is recommended to be integrated into the review and discussion step (in purple) (Source: [4]).

2.7.2 Actions

Automation plays a very important part in improving collaborative software development. GitHub Actions enables developers to create custom software development life-cycle workflows directly from the GitHub repository. Since it's fully integrated into GitHub, a developer can use this feature to perform any automated job at any stage of GitHub workflow. It can perform changes on the code and even collaborate over pull request and issues, including Continuous Integration (CI).

The basic notion of Continuous Integration is that all work made by a team is continually compiled, built and tested [57]. CI on GitHub is merely the most used implementation of the GitHub Actions for automation processing and also the most documented one.

One study [58] has shown that teams working on CI-enabled repositories on GitHub are significantly more effective at merging pull requests and are considerably more able to discover bugs than teams not using CI. Which suggests that CI improves productivity without having a negative effect on code quality.

Another study [59] showed that CI saves on average one review comment per pull request, that the changes induced requests by developers decrease while the number of changes to the pull request code does not decrease. In our work one changed line of code can be the difference between a well established and reliable protocol on the Ethereum network and millions of US dollars lost due to a bug in the code.

GitHub Actions are event driven, meaning a developer can specify a series of commands to be run after a specific event has occurred. This gives a developer the ultimate freedom to customize work and automate tasks all along the software development life-cycle. To be more specific, we will now present an overview on how it works.

GitHub Actions uses the YAML syntax to define events, jobs and steps. These YAML files are called Workflows and are stored in a project's repository, inside the directory "`.github/workflows`". A project can have any number of YAML workflows. A visual example of a GitHub's YAML workflow is shown in Section 3.2 specified by the following attributes:

Workflow Is an automated procedure that any developer can add to their repository. It can have one or more jobs and is triggered by an event. The workflow can be used from building, to testing, to packaging, to releasing and even to deploying a GitHub project.

name Is the workflow's name which will appear on the GitHub's Action tab. Is optional.

on Specifies the Event which is a determined activity that will trigger a workflow. It can be specified to run on certain tags, branches or paths.[20]

jobs Groups together all will run on that workflow. A job is the set of steps to be executed. If a workflow has multiple jobs, these will run in parallel by default.

runs-on Configures the job to be run that specific machine. The self-hosted means GitHub will find a machine matching the job's label.

steps Is an individual task which aggregates commands running inside a job. It can be either a shell command or an action and it can share data with different steps inside the same job.

uses Is a standalone command. An action can be created or called from existing ones on the GitHub community. In the example below, the *actions/checkout@v2* is telling GitHub to check out the repository and download it to the runner, allowing the runner to perform actions against the code.

run This keyword will tell GitHub to execute the command directly on the runner's shell.

2.7.3 Marketplace

GitHub Marketplace[21] is an online platform where you can share your apps and actions with the rest of the GitHub community. This provides a convenient way to add SmartBugs as an automated action to every Solidity developers' repository.

In order to publish a GitHub Action on the Martketplace the following criteria must be met:

[20] Full list of events: `https://docs.github.com/en/actions/learn-github-actions/events-that-trigger-workflows`
[21] GitHub Marketplace: `https://github.com/marketplace`

- The action must provide value to the GitHub community.
- It must not persuade users away from GitHub.
- It has to include valid contact information for the publisher.

3 Implementation

Contents

3.1 Motivation 37
3.2 SARIF Converters 38
3.3 SASP 43
3.4 GitHub Review Bot 45
3.5 SmartBugs 49

In this chapter we present our improvements on the SmartBugs framework. We start by presenting our system's architecture. We then describe our implementation for the SARIF converters and how to replicate them for other tools. We overview our server solution where clients can request a SmartBugs analysis with a SARIF output. Furthermore, we present our chosen consumer, GitHub, and the advantages and disadvantages of its two adapters, for running an automated analysis on the user's repository. Finally, we briefly discuss the SmartBugs' changes and added features.

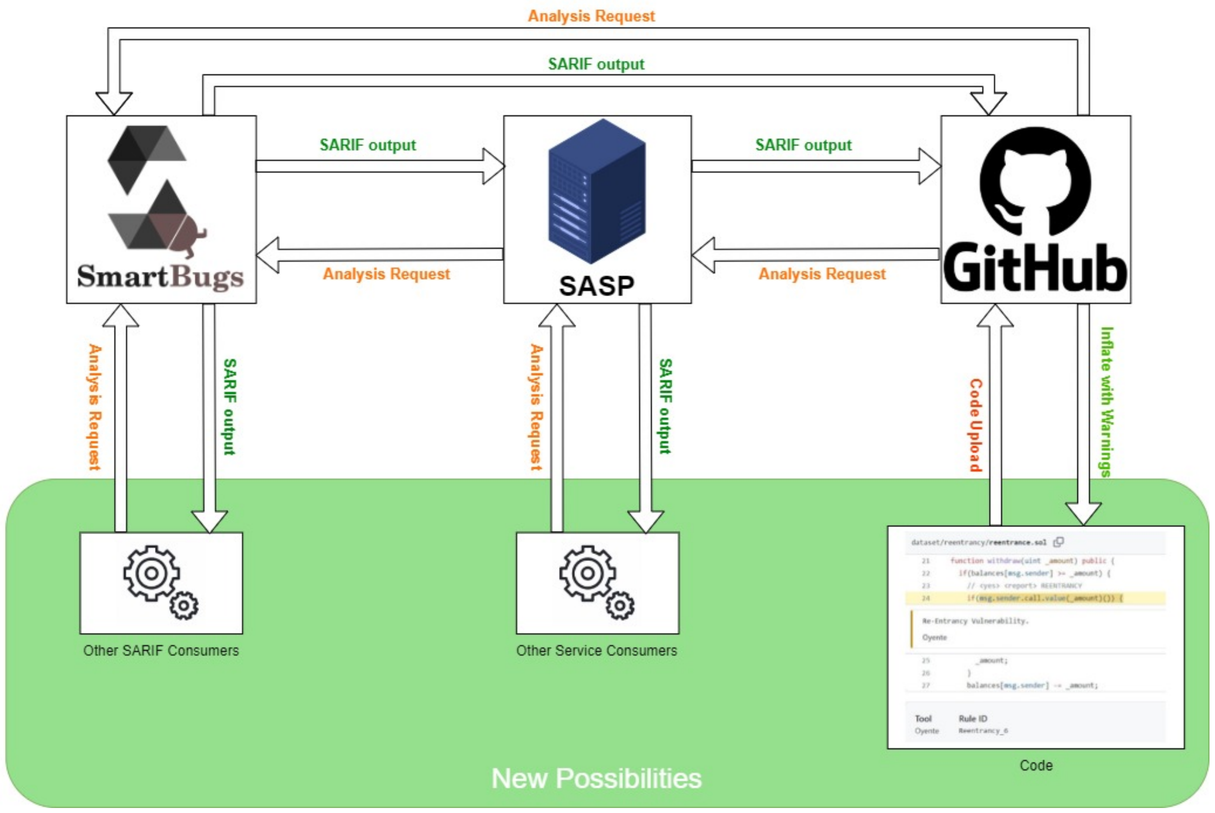
3.1 Motivation

Even though SmartBugs incredibly eases the research and reproduction of new static analysis tools, there is still a big community of developers left outside of SmartBugs' scope, the smart contract developers. While using SmartBugs for reviewing their contracts, a developer tends to find himself with a huge overhead on parsing the outputs, unifying the results and acknowledge the information produced by its tools. Apart from missing an option to discard duplicate results and false positives, SmartBugs also lacks an easy and intuitive interface to display its results.

Some developers also lack the necessary computer power to run the analysis. Or perhaps don't want to install the requirements to run SmartBugs as they're still required to install Python, Docker and all the mandatory modules. If a developer has all the conditions stated just now, it still needs to manually run the analysis himself. What doesn't feel like a big burden, could be turned into something automatic, like a step in the continuous integration process, which means one less detail that the developer needs to worry about.

We introduce SmartBugs as a Platform-as-Service (PaS) to automate and increase the usage of static analysis tools from smart contract developers. SmartBugs now includes the possibility of displaying the results of all analyses executed in an unified single file, formatted in the same standard, for all of its tools. Furthermore, it provides two plugins for one end consumer where the developer is only required to submit the code and SmartBugs' automation handles the rest. Moreover we have added Conkas, a new promising tool, to its framework that is very low time consuming with great accuracy metrics.

Our developed system's architecture can be seen in Figure 3.1. Basically, we have SmartBugs that can now be integrated with other tools that consume SARIF. We then have the SASP service who is responsible for executing the analysis, whether requested by GitHub or any other service consumer. Finally, we have GitHub, our chosen end consumer, which can request an automatic SmartBugs analysis either directly on a GitHub allocated server or through the SASP service. GitHub will then inflate the repository code with the analysis results.



```
dataset/reentrancy/reentrance.sol
  function withdraw(uint _amount) public {
    if(balances[msg.sender] >= _amount) {
      // <yes> <report> REENTRANCY
      if(msg.sender.call.value(_amount)()) {
Re-Entrancy Vulnerability.
Oyente
        _amount;
      }
      balances[msg.sender] -= _amount;
```

Tool	Rule ID
Oyente	Reentrancy_6

Figure 3.1: Overview of the final implementation. This concept can be applied to any other static analysis tool.

We hope with this work to bring out this standardized methodologies to the attention of all static analysis developers in order to increase their adoptions. We also describe how they can replicate our work to help build safer and more secure coding practices.

3.2 SARIF Converters

The SARIF converters for each tool were written in Python based on the *sarif_om*[1] module developed by Microsoft on GitHub. This module contains classes for the object model presented by the Static Analysis Results Interchange Format (SARIF) Version 2.1.0 file format, an OASIS Committee Specification, which we reviewed in Chapter 2.5.

When running SmartBugs' analysis on a repository, it provides three options for generating the SARIF output: Creating an independent file for each of the files analysed, aggregating the SARIF results by tool and generating a unique file containing all results of the analysis. Most consumers request the last one as all results of an analysis must be together in a single file.

[1] GitHub sarif_om module: https://github.com/microsoft/sarif-python-om

Each tool's parser mechanism was developed with a plugin design pattern, where each converter has its own Python file, to ease the addition of converters for other tools. These can be found in the `smartBugs/scr/output_parser/` folder[2]. Furthermore we developed a sarif converter from sarif_om to SARIF JSON format which had been requested by multiple users on GitHub, with no statement from SARIF's creator on how to achieve this conversion.

As we have seen in Chapter 2.5, SARIF's standard can be pretty complex and confusing. In order to make not only our converters' code easier to understand, but also to make adding another converter a cleaner task, we have created a useful class, the *SarifHolder*, and some parsing functions for SARIF. They simplify the construction of SARIF outputs by reducing the number of fields that a developer has to research while programming the tool. These functions were developed based on a balance from our research, SmartBugs tools' outputs and GitHub's SARIF consumer parameters. They should not be regarded as definitive nor exhaustive as they do not reflect all possible details on this standard. However they manage to agglomerate all fields outputted by every tool added so far and thus are deemed comprehensive enough for our work.

In Figure 3.2 we can see a diagram representation of our SARIF construction. The title of each concept represents the objects name and in the case of the *parse* represents one of the simplified functions. In red we have the return object type of each concept. Below we specify the attributes of each object and their type, using **bold** to define the mandatory ones.

With this diagram we hope to facilitate the re-usage of our code so that more and more static analysis developers adopt SARIF's standard. Not only in a comprehensive Solidity environment but also for all purpose programming.

[2]SmartBugs SARIF converters on GitHub: https://github.com/smartbugs/smartbugs/tree/master/src/output_parser

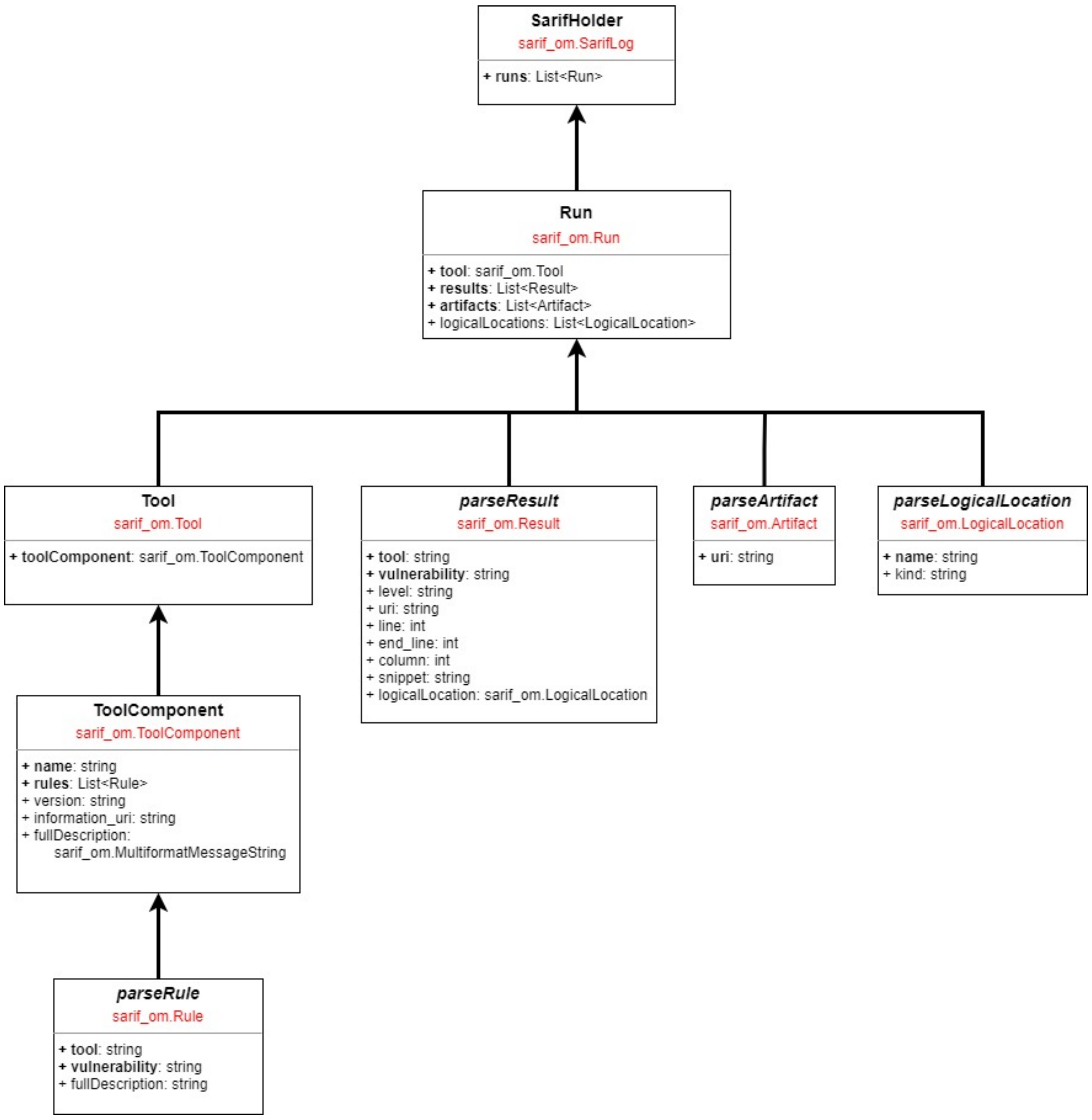


Figure 3.2: SarifHolder Construction Design - A simplified SARIF version ready to be consumed

Due to the scarcity of output information generated by the tools regarding the environment in which they were run, we opted to omit the *invocations* key. Even though it collects fairly important content for the analysis, most consumers end up generating *invocations*' fields themselves. For example, GitHub only displays the time of the analysis sub-field, and when it's missing it is automatically replaced with the time when the SARIF file was received by the consumer. This small difference in timestamp windows was regarded as irrelevant due to having to effect in any other part of the analysis.

For a comparison with the examples from 2.5.2 we provide the actual SARIF output using one of our converters for the analysis of the contract SimpleDao completed by the Oyente [16] tool in Listing 3.1.

Listing 3.1: SARIF output produced by Oyente analysing SimpleDAO

```
{
    "runs": [
    {
        "tool": {
            "driver": {
                "name": "Oyente",
                "fullDescription": {
                    "text": "Oyente runs on symbolic execution, determines which
                        inputs cause which program branches to execute, to find
                        potential security vulnerabilities. Oyente works directly with
                         EVM bytecode without access high level representation and
                        does not provide soundness nor completeness."
                },
                "informationUri": "https://oyente.tech/",
                "rules": [
                {
                    "id": "DenialService_2",
                    "name": "DenialServiceVulnerability",
                    "shortDescription": {
                        "text": "Callstack Depth Attack Vulnerability."
                    }
                },
                {
                    "id": "Reentrancy_6",
                    "name": "ReentrancyVulnerability",
                    "shortDescription": {
                        "text": "Re-Entrancy Vulnerability."
                    }
                }
                ],
                "version": "0.4.25"
            }
        },
        "artifacts": [
        {
            "location": {
                "uri": "dataset/reentrancy/simple_dao.sol"
            },
            "sourceLanguage": "Solidity"
        }
        ],
        "logicalLocations": [
        {
            "kind": "contract",
            "name": "SimpleDAO"
        }
        ],
        "results": [
        {
            "message": {
                "text": "Callstack Depth Attack Vulnerability."
            },
            "level": "warning",
            "locations": [
            {
                "physicalLocation": {
                    "artifactLocation": {
                        "uri": "dataset/reentrancy/simple_dao.sol"
                    },
                    "region": {
                        "startColumn": 18,
                        "startLine": 19
                    }
                }
            }
            ],
            "ruleId": "DenialService_2"
        },
        {
            "message": {
                "text": "Re-Entrancy Vulnerability."
            },
            "level": "warning",
            "locations": [
            {
                "physicalLocation": {
                    "artifactLocation": {
                        "uri": "dataset/reentrancy/simple_dao.sol"
                    },
                    "region": {
                        "startColumn": 18,
                        "startLine": 19
                    }
                }
            }
```

```
                ],
                "ruleId": "Reentrancy_6"
            }
            ]
        }
    ],
    "version": "2.1.0",
    "$schema": "https://raw.githubusercontent.com/oasis-tcs/sarif-spec/master/
        Schemata/sarif-schema-2.1.0.json"
}
```

3.2.1 SARIF's Vulnerabilities Mapping Table

Since SARIF requires more detailed information than what most tools generate, we created a table for mapping all vulnerabilities found. This table serves two purposes: Identify each vulnerability and inflate the SARIF output with extra required information about what was reported; And facilitate the addition of new SARIF keys in order to produce a more comprehensive report in the future.

At the moment, this table is used to inflate the *ruleId* property and the *name* key. For the first one it is critical to be unique for each of the vulnerabilities reported regardless of the tool. We created each ruleId by concatenating the vulnerability type with an incrementing counter, however this is merely a suggestion, it can be anything as long as it is unique in the table. Finally, the *name* key is inflated with the category type following the DASP-10 taxonomy, for each of the reported rules.

This table can be found in the `smartBugs/src/output_parse/sarif_vulnerability_mapping.csv` file. The vulnerabilities reported by Oyente can be seen mapped in Table 3.1.

Tool	RuleId	Vulnerability	Type
oyente	Arithmetic_11	Integer Overflow.	Arithmetic
oyente	Arithmetic_12	Integer Underflow.	Arithmetic
oyente	AccessControl_16	Parity Multisig Bug 2.	AccessControl
oyente	DenialService_2	Callstack Depth Attack Vulnerability.	DenialService
oyente	Reentrancy_6	Re-Entrancy Vulnerability.	Reentrancy
oyente	TimeManipulation_3	Timestamp Dependency.	TimeManipulation

Table 3.1: SARIF Vulnerability Mapping Table for Oyente

3.3 SASP

We build SASP, as we have seen in Chapter 2.6, to work as a server for communicating the static analysis tools' results to their end consumers. Basically, SASP's objective is to connect static analysis tools and development platforms via itself in order to combine all analysis tool outputs and provide a standardized results output that can be consumed by the developer tool that initiated the request.

On a practical level our objective for SASP is when a developer submits the code, the end consumer automatically sends an analysis request to SASP who is then going to initiate an analysis for the submitted code. This analysis' result will be returned to the end consumer who will then notify the programmer of any reported vulnerabilities found. A visual representation of SASP can be seen in Figure 3.3.

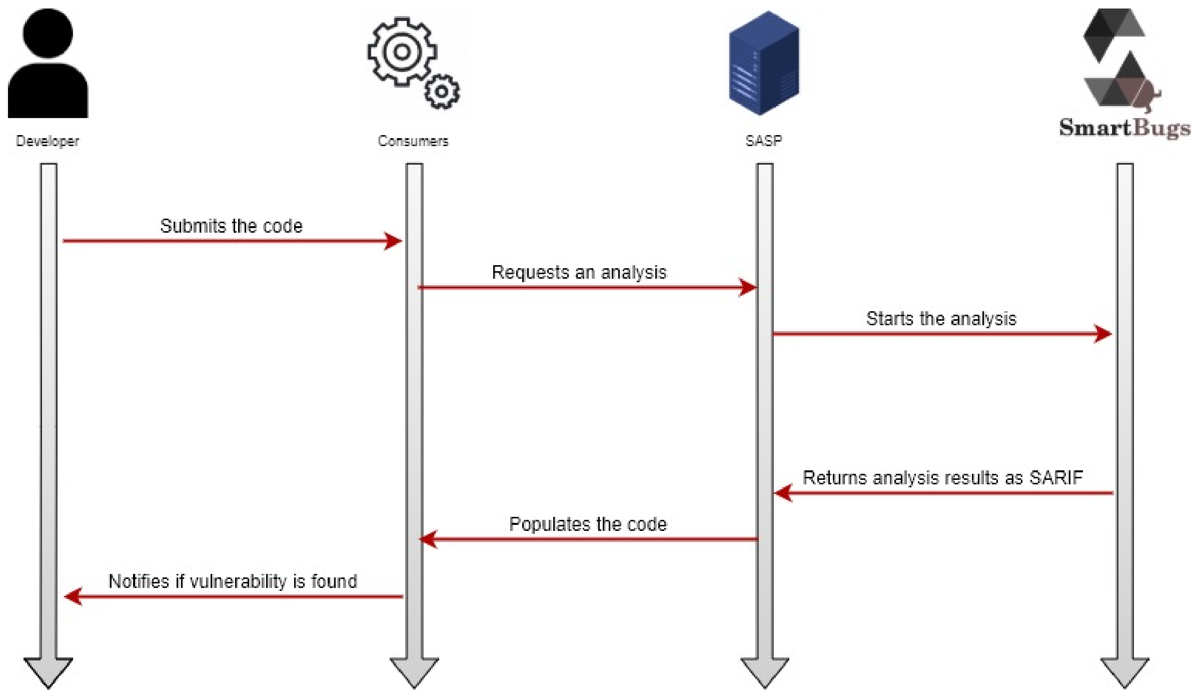


Figure 3.3: Example of a end consumer working with SASP to populate comments for an arbitrary code submitting

SASP was developed using Python[3] for compatibility reasons with SmartBugs, with the Flask module. Its code is a single Python file that requires SmartBugs repository installed in the computer and declared in the $\$PYTHONPATH$ environment variable. We chose the Flask module due to its simplicity and automatic thread handling when receiving requests from multiple users.

Even though there is no formal specification for the SASP, we followed these considerations [49]:

- The protocol is HTTP based.
- It has a secure authentication mechanisms based on a token auth.
- Sessions and state are present as the user repository gets stored in the server based on the token authentication.

[3]SASP repository on GitHub: `https://github.com/dindonero/smartbugs-sasp`

- Results from static analysis tools may be vast, so the protocol supports streaming the results. This is directly supported by Python and Flasks underlying mechanics.

- Fault tolerance is very important so dropped connections, timeouts, authentication or authorization failures, badly formed requests and so forth, are addressed.

As good code practice states, we should make one unique version of SmartBugs for all of its usage. In order to achieve this, we must use the same code for running SmartBugs locally and for executing it on the SASP server. This quickly presented a problem. The way that SmartBugs works is by reading the code files that is going to analyze directly from the disk, which when running in a web-server allows for a Directory Traversal Attack. This exploit is a web security vulnerability (often referred to as file path traversal) that allows an attacker to read arbitrary files on a server that is executing an application. This files could contain application code and data, back-end system credentials, and critical operating system files. In SmartBugs' situation, an attacker was even able to write to arbitrary files on the server, allowing them to change application data or behavior which could eventually lead them to gain complete control of the server. Our solution was based on the mitigation provided by [60] in `II.C.1.` since we are working with an HTTP server and where, in our specific case, path strings starting from the root directory are not a concern.

3.4 GitHub Review Bot

As we intend to entice developers on a more frequent use of the static analysis tools, consequentially resulting in less vulnerable programs (i.e. smart contracts) being deployed, we provide a complete automatic setup on top of one selected end consumer. And so, in contemplation of providing a mechanism for a straightforward interpretation of the results produced by these tools, we created an automated GitHub review bot[4] using GitHub's SARIF processing feature to intuitively display the analysis results as inflated comments in the code.

The automated analysis plugin is written in YAML for GitHub Actions, as we have seen in Chapter 2.7.2. Our Action takes as input the tools that the user wants to run in the analysis and the generated token *github.sha*, unique to each GitHub account. The user must bear in mind that the size of the repository and the number of tools chosen will exponentially increase the cost in computation time, so we recommend choosing at most 3 tools. Later on, and depending on the adoption of our system, we might change the GitHub's generated token to a paid token where only subscribing users get access to this premium system (while the other will have only the free version which we will see in 3.4.1).

[4]SASP GitHub Adapter: https://github.com/dindonero/smartbugs-static-analysis/

In order to run the SASP analysis, a user must add a *.yml* file to its repository in the `.github/worflows` and specify when to trigger the analysis and the tools to be executed. A user can specify the action to start based on a schedule event (based on time, i.e. run every week), a manual event (not automatic) and/or a triggering event, e.g. *pull_request* when a pull request is created or *push* every time code is pushed into the repository. An example for this config file can be seen in Listing 3.2. The full list of events and their descriptions can be found here: https://docs.github.com/en/actions/learn-github-actions/events-that-trigger-workflows

This action will then execute, in the user's repository where it was added, the code deployed in the action present in `dindonero/smartbugs-static-analysis@v1.0`. This is executed in a GitHub hosted machine.

Our action's implementation is based on executing a versatile Python file that reads the entire repository and uploads it to our SASP server. This design was selected so that it could be executed independently of the GitHub environment. Any IDE or consumer that wants to use our service is able to do so by simply running the Python code in our repository or by directly making a request to our server.

The request must be HTTP POST to http://perseus.inesc-id.pt:8000/. Its body must be composed by an unique identifiable string, the "user-hash" key, and a list of strings describing the tools to be executed, the "tools" key. The files to be uploaded must be appended in the *multipart/form-data* part of the request.

Listing 3.2: Example of the file *.github/workflows/smartbugs.yml* that triggers the SmartBugs static analysis automatically at every pull request

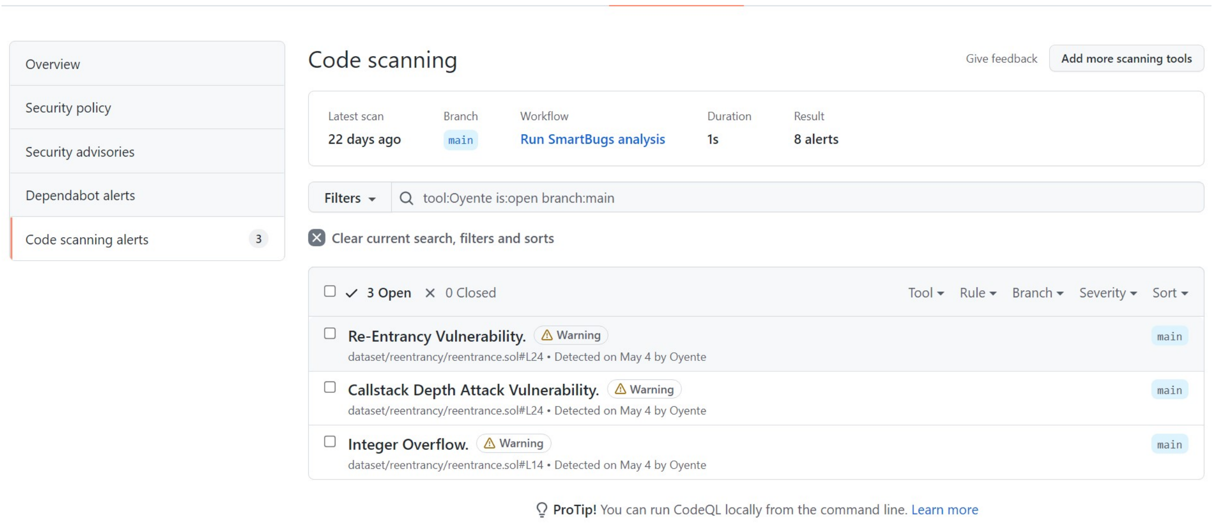
```yaml
name: "Run SmartBugs static analysis on this repository"

# Specify when the workflow is run
on:
    pull_request:

jobs:
    deployment:
        runs-on: ubuntu-latest
        steps:
            - name: Run SmartBugs Static Analysis
            uses: dindonero/smartbugs-static-analysis@v1.0
            with:
                # Specify the tools to be executed
```

```
                    tool: 'conkas oyente mythril'
```

The result of a successful analysis reported by GitHub executed on the SimpleDAO contract by Oyente can be seen in Figures 3.4 and 3.5



Overview
Security policy
Security advisories
Dependabot alerts
Code scanning alerts 3

Code scanning

Give feedback | Add more scanning tools

Latest scan	Branch	Workflow	Duration	Result
22 days ago	main	Run SmartBugs analysis	1s	8 alerts

Filters | tool:Oyente is:open branch:main

Clear current search, filters and sorts

✓ 3 Open ✕ 0 Closed — Tool, Rule, Branch, Severity, Sort

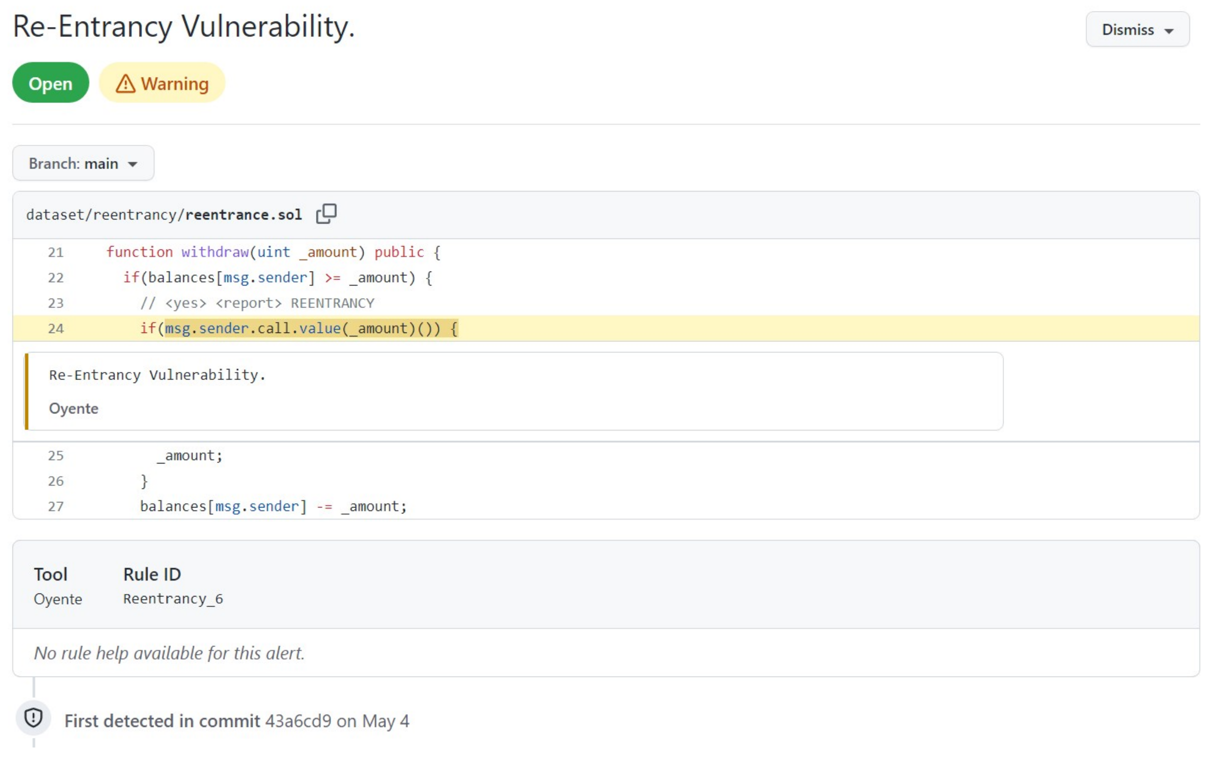
Re-Entrancy Vulnerability. ⚠ Warning — main
dataset/reentrancy/reentrance.sol#L24 • Detected on May 4 by Oyente

Callstack Depth Attack Vulnerability. ⚠ Warning — main
dataset/reentrancy/reentrance.sol#L24 • Detected on May 4 by Oyente

Integer Overflow. ⚠ Warning — main
dataset/reentrancy/reentrance.sol#L14 • Detected on May 4 by Oyente

ProTip! You can run CodeQL locally from the command line. Learn more

Figure 3.4: GitHub displaying the vulnerabilities reported by Oyente on the SimpleDAO contract



Re-Entrancy Vulnerability.

Open | ⚠ Warning | Dismiss

Branch: main

dataset/reentrancy/reentrance.sol

```
function withdraw(uint _amount) public {
  if(balances[msg.sender] >= _amount) {
    // <yes> <report> REENTRANCY
    if(msg.sender.call.value(_amount)()) {
```

Re-Entrancy Vulnerability.

Oyente

```
      _amount;
    }
    balances[msg.sender] -= _amount;
```

Tool	Rule ID
Oyente	Reentrancy_6

No rule help available for this alert.

First detected in commit 43a6cd9 on May 4

Figure 3.5: GitHub inflating the SimpleDAO contract with the Reentrancy Vulnerability reported by Oyente

3.4.1 NoSASP approach

Alternatively to the SASP approach[5] that we have seen above, we also developed a server-free version of the automated analysis for GitHub. This approach is independent of our service and it runs SmartBugs directly on a free-of-use GitHub hosted machine.

Even though this version of our work is more scalable, it presents other constraints. The GitHub machine comes with Python installed but every time it runs an analysis it requires the download of: All the Python modules required to run SmartBugs, the entire SmartBugs repository (which includes $SB^{CURATED}$), and every tool requested by the developer. Besides this, the machine is limited by the following specs:

- 2-core CPU
- 7 GB of RAM memory
- 14 GB of SSD disk space

The amount of memory available is bounding for everything done in the machine. So the total sum of the SmartBugs repository's size, the downloaded tools' size and the user repository's size cannot surpass 14 GBs of memory. Considering that all 11 tools represent about 20 GBs of memory this limitation is capable of affecting a big number of users.

3.4.2 False positives

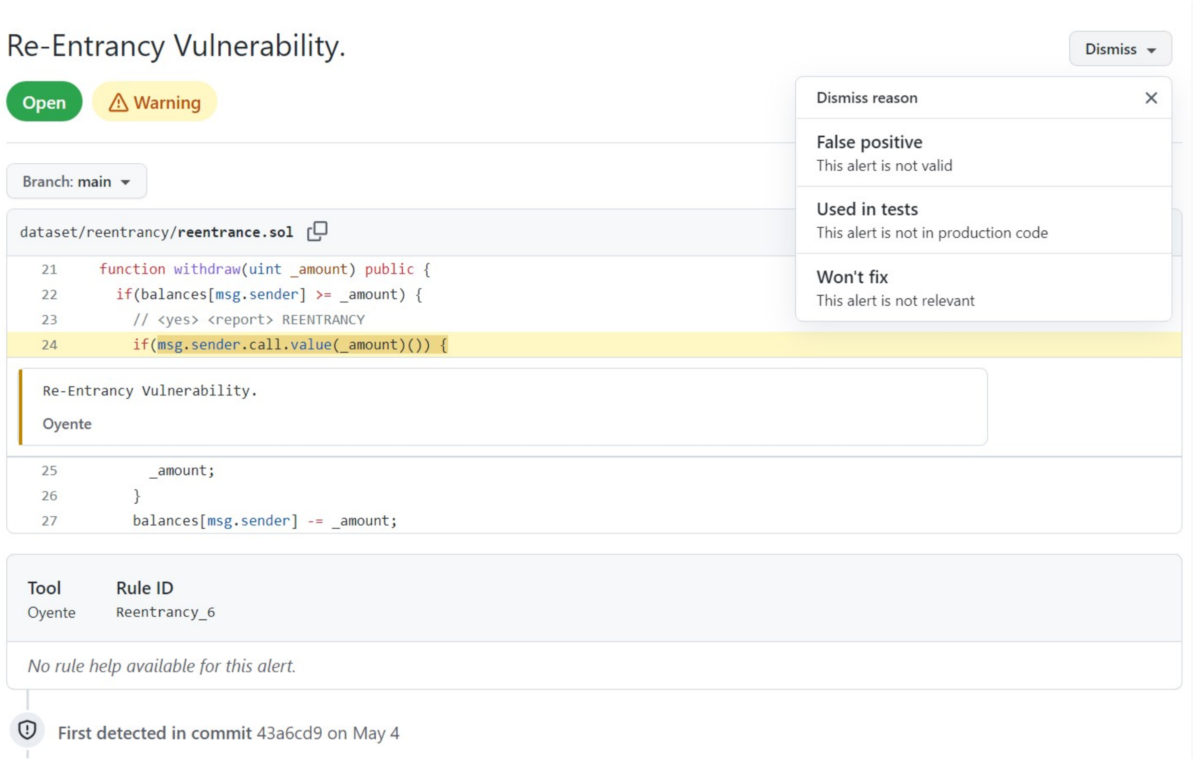
We have seen in the studies presented in Chapter 2.4.4 that static analysis tools vigorously try to eradicate false negatives. Conversely, their developers also strive to minimize the amount of false positives reported by their tools. The false positive vulnerabilities problem leads to one of the main reasons why programmers dismiss using static analysis tools in their work. Because besides having to sort through piles of logging output, they still need to filter and distinguish the real code vulnerabilities from the noise.

GitHub's SARIF interpretation framework supports a straightforward mechanism to tag every vulnerability found so that it can be ignored the next time it is reported again. This framework provides the following three tagging options: False positive, Used in tests and Won't fix. And they're ignored by a simple click of a button. The dismissal options box can be seen in Figure 3.6.

The GitHub's framework false vulnerability report detects a vulnerability as duplicate if the combination of the repository, the file path, and the SARIF result data is the same. In a future pull requests where the same vulnerability is reported, instead of warning the developer, GitHub will mark it as rediscovered. If the developer had tagged it with one of the three options, GitHub will simply discard the vulnerability

[5]NoSASP GitHub Adapter: https://github.com/dindonero/smartbugs-local-static-analysis

report. The information about the results is stored in the GitHub's SARIF API endpoint. We hope to provide an more embraceable methodology to ignore unwanted results so that a developer can focus on its code real security problems.


Re-Entrancy Vulnerability.

dataset/reentrancy/reentrance.sol

```
function withdraw(uint _amount) public {
  if(balances[msg.sender] >= _amount) {
    // <yes> <report> REENTRANCY
    if(msg.sender.call.value(_amount)()) {
```

Re-Entrancy Vulnerability.

Oyente

```
      _amount;
    }
    balances[msg.sender] -= _amount;
```

Tool	Rule ID
Oyente	Reentrancy_6

No rule help available for this alert.

First detected in commit 43a6cd9 on May 4

Figure 3.6: GitHub option to report false positives

3.5 SmartBugs

3.5.1 Conkas

Conkas[6] is a symbolic execution-based static analysis tool for the EVM. It can examine Solidity-based Ethereum Smart Contracts as well as the compiled runtime byte-code. Due to being a modular tool, anyone can easily add their own modules to analyze specific vulnerabilities. It uses Z3 as the SMT Solver and Rattle as the Intermediate Representation (IR).

Conkas has 5 modules to detect different categories of vulnerabilities. These categories are based on the DASP 10, which we reviewed under Chapter 2.3, and are the following:

1. Arithmetic

2. Reentrancy

[6]Conkas GitHub: https://github.com/nveloso/conkas

3. Time Manipulation
4. Transaction Ordering Dependence
5. Unchecked Low-Level Calls

This tool was added to SmartBugs through a contribution from its author. Furthermore we designed Conkas' SARIF converter and added it to our SASP framework. Conkas can now be accessed like any other SmartBugs' tool through our service. He also benefited from SmartBugs' framework to compare Conkas performance against all the other tools. This tool shows great promise as it is one of the less CPU consuming tools, with great accuracy metrics. Meaning it can generate accurate results without taking up a lot of time. We can see the execution time for Conkas and the other 9 tools in Table 3.2. And the accuracy statistics in Table 3.3.

#	Tool	Avg. Execution Time	Total Execution Time
1	Conkas	0:00:32	1:14:37
2	Honeybadger	0:01:12	2:49:03
3	Maian	0:03:47	8:52:25
4	Manticore	0:12:53	1 day, 6:15:28
5	Mythril	0:00:58	2:16:21
6	Osiris	0:00:21	0:50:25
7	Oyente	0:00:05	0:12:35
8	Securify	0:02:06	4:56:13
9	Slither	0:00:04	0:09:56
10	Smartcheck	0:00:15	0:35:23

Table 3.2: Conkas Execution Time Statistics (Source: [7])

Category	Conkas	Honeybadger	Maian	Manticore	Mythril	Osiris	Oyente	Securify	Slither	Smartcheck	Total
Access Control	0/24 0%	0/24 0%	0/24 0%	5/24 21%	4/24 17%	0/24 0%	0/24 0%	1/24 4%	6/24 25%	2/24 8%	8/24 33%
Arithmetic	19/23 83%	0/23 0%	0/23 0%	13/23 57%	16/23 70%	13/23 57%	18/23 78%	0/23 0%	0/23 0%	1/23 4%	22/23 96%
Denial Service	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	0/14 0%	1/14 7%	1/14 7%
Front Running	2/7 29%	0/7 0%	0/7 0%	0/7 0%	2/7 29%	0/7 0%	2/7 29%	2/7 29%	0/7 0%	0/7 0%	2/ 7 29%
Reentrancy	30/34 88%	19/34 56%	0/34 0%	15/34 44%	25/34 74%	21/34 62%	28/34 82%	14/34 41%	33/34 97%	30/34 88%	33/34 97%
Time Manipulation	7/7 100%	0/7 0%	0/7 0%	4/7 57%	0/7 0%	2/7 29%	0/7 0%	0/7 0%	3/7 43%	2/7 29%	7/ 7 100%
Unchecked Low Calls	62/75 83%	0/75 0%	0/75 0%	9/75 12%	60/75 80%	0/75 0%	0/75 0%	50/75 67%	51/75 68%	61/75 81%	70/75 93%
Other	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/5 0%	0/ 5 0%
Total	120/224 54%	19/224 8%	0/224 0%	46/224 21%	107/224 48%	36/224 16%	48/224 21%	67/224 30%	93/224 42%	97/224 43%	143/224 64%

Table 3.3: Conkas Accuracy Metrics (Source: [7])

3.5.2 Unit Tests

In order to ensure that any future upgrades to SmartBugs don't unwillingly break functionality to our adapters, we also produced nine basic unit tests that verify the SARIF functionality. When executed, they run different tools against the SimpleDAO contract and compare the actual SARIF output to the expected output.

3.5.3 System Requirements

SmartBugs 2.0 requires Docker and Python3 installed with the modules PyYAML, solidity parser, docker, GitPython, sarif-om, attrs and pandas.

In order to run SASP, Flask module with the version 1.1.2 or higher is also necessary. A user who prefers to run our server or someone who intends to run the tests, needs to add the root folder of SmartBugs' repository to the $\$PYTHONPATH$ environment variable.

3.5.4 Updated CLI

SmartBugs provides a command line interface that allows users to access the datasets and execute various analysis tools on smart contracts.

SmartBugs updated command line interface can be invoked as:

```
    smartBugs.py    [-h, --help]
                    (--file FILES | --dataset DATASETS)
                    --tool TOOLS
                    --info TOOLS                  Information about the tool
                    --skip-existing               Skip analyses that already have results
                    --processes PROCESSES         Number of parallel execution
                    --list tools datasets         List tools or datasets
                    --import-path IMPORT_PATH     Imports project's root directory
                    --output-version {v1,v2,all}  Specifies output - v1:Json; v2:SARIF
                    --aggregate-sarif             Aggregate SARIF outputs by tool
                    --unique-sarif-output         Aggregates all SARIF analysis outputs in
                         a single file
```

3.5.5 Updated Methodology for adding tools

The process of adding tools to SmartBugs is designed to be straightforward and practical, allowing the user to determine how the tools are executed based on their needs. All 11 SmartBugs tools use Docker images downloaded from the Docker Hub.

The tools' description is stored in each tool plugin. The plugin defines the Docker image's name, the tool's name, the command line to execute the tool, and, ideally, the tool's description. Once a docker image for the tool is available, a user can add it to SmartBugs by creating a new YAML configuration file inside the `smartBugs/config/tools/` folder. The specification of that file should follow the Conkas example in Listing 3.5.5.

```
docker_image:
    default: nveloso/conkas
cmd: -fav -s
info: Conkas is based on symbolic execution, determines which inputs
    cause which program branches to execute, to find potential
    security vulnerabilities. Conkas uses rattle to lift bytecode to a
     high level representation.
```

SmartBugs automatically pulls the results from the tool's printed output. Conversely, if a tool saves the analysis result in the docker image, the location to that file should be specified in the configuration file by adding an extra field, as the example illustrated in Listing 3.5.5.

```
output_in_files:
    folder: /result/output.json
```

This configuration is fulfilling enough for any tool a user might want to add to the SmartBugs' framework. Nevertheless, if the objective also comprises of enabling the tool into the SmartBugs 2.0 adapter environment, a developer is additionally required to create a SARIF converter for his tool. The SARIF converter must be a new Python file in the `smartBugs/src/output_parser/` folder, preferably named after the tool. Inside the file there should be a function called *parseSarif* which receives the original output as an argument and returns a SARIF Run object. The function must reflect the conversion of its original output to SARIF. We recommend following the methodology presented in 3.2 where the developer is additionally able to look at the other converters as an example, such as the Maian converter in Listing 3.5.5.

Furthermore, the developer must add all possible reported vulnerabilities from his new tool into the sarif_vulnerability_mapping.csv table and fill all necessary information, which we reviewed under Section 3.2.1. He is also presented with the choice of adding extra information not present in the columns composing the table.

```
def parseSarif(self, maian_output_results, file_path_in_repo):
    resultsList = []
    rulesList = []

    for vulnerability in maian_output_results["analysis"].keys():
        if maian_output_results["analysis"][vulnerability]:
            rule = parseRule(tool="maian", vulnerability=vulnerability)
            result = parseResult(tool="maian", vulnerability=vulnerability,
                level="error", uri=file_path_in_repo)

            rulesList.append(rule)
            resultsList.append(result)

    artifact = parseArtifact(uri=file_path_in_repo)

    tool = Tool(driver=ToolComponent(name="Maian", version="5.10", rules=
        rulesList, information_uri="https://github.com/ivicanikolicsg/MAIAN",
         full_description=MultiformatMessageString(text="Maian is a tool for
        automatic detection of buggy Ethereum smart contracts of three
        different types prodigal, suicidal and greedy.")))

    run = Run(tool=tool, artifacts=[artifact], results=resultsList)

    return run
```

4

Evaluation

Contents

4.1 SARIF converters 57

4.2 SASP Service 61

4.3 GitHub Adapter 64

In this chapter we discuss the evaluation's results of our improvements to the tool presented in Section 2.4. Our evaluation consists mainly on manual verifications.

Our work validation will consist in three steps. Checking the validity of our SARIF converter, verifying the functionality of our SASP service and confirm the completeness of our GitHub adapter feature. For each of the stages we will make use of the dataset $SB^{CURATED}$ incorporated in the SmartBugs environment to the extent of verifying each step capabilities and correctness.

4.1 SARIF converters

This section is the most complex of our work and as such is divided in three subsections. The converter execution, the SARIF standard validation and the vulnerabilities correctness verification.

4.1.1 Converter Execution

While some tools have the same static output, meaning that the output is always represented by the same fields, others present a more dynamic one, where the output in some cases may not contain certain fields and those fields may appear in different orders. On both output types, but specially when converting a dynamic one, we noticed some cases where the vulnerabilities were reported defectively. The vulnerabilities that fall into this case are discarded when converting to SARIF.

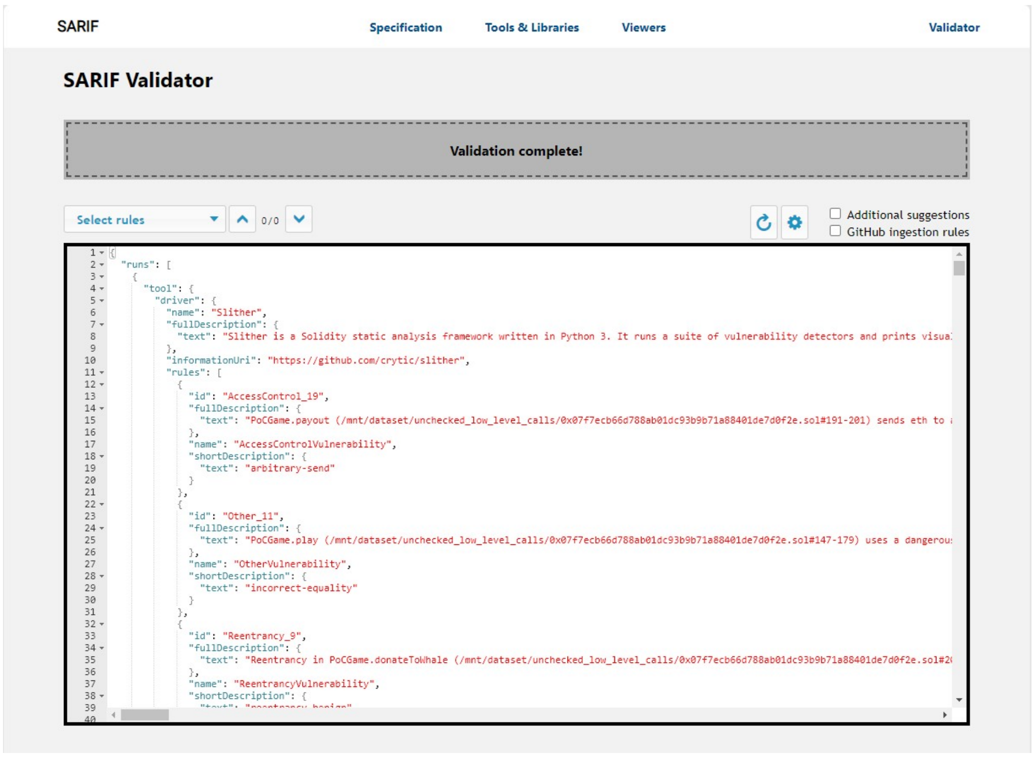
In this part of our work, the entire $SB^{CURATED}$ dataset was used as an accuracy metric for testing corner cases on our converters. Since we don't catch exceptions thrown by translation errors, we took advantage of that in order to acknowledge faulty converters. Every tool converter against every smart contract present in the dataset was executed and fixed until all of them positively pass this stage.

We feel pretty confident that as long as a converter doesn't fail or crash, it is working correctly and every eligible reported vulnerability for translation is being handled correctly. Nevertheless, we assure their correctness in the next steps of the evaluation.

4.1.2 SARIF Standard Validation

In this step, we used the public free SARIF validation tool provided by Microsoft[1] to ensure our standard correctness. A correctly formatted document can be seen in Figure 4.1 while an older badly formatted one can be seen in Figure 4.2.

[1]SARIF Validation Tool: https://sarifweb.azurewebsites.net/Validation

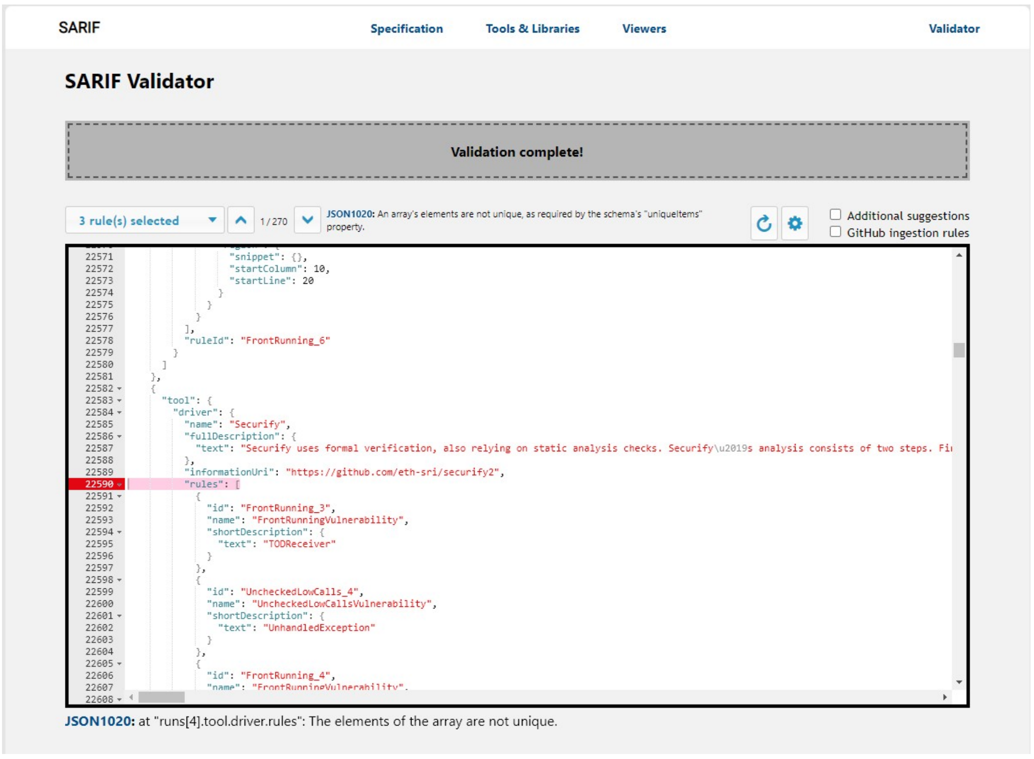


SARIF Validator

Validation complete!

```
{
  "runs": [
    {
      "tool": {
        "driver": {
          "name": "Slither",
          "fullDescription": {
            "text": "Slither is a Solidity static analysis framework written in Python 3. It runs a suite of vulnerability detectors and prints visua
          },
          "informationUri": "https://github.com/crytic/slither",
          "rules": [
            {
              "id": "AccessControl_19",
              "fullDescription": {
                "text": "PoCGame.payout (/mnt/dataset/unchecked_low_level_calls/0x07f7ecb66d788ab01dc93b9b71a88401de7d0f2e.sol#191-201) sends eth to a
              },
              "name": "AccessControlVulnerability",
              "shortDescription": {
                "text": "arbitrary-send"
              }
            },
            {
              "id": "Other_11",
              "fullDescription": {
                "text": "PoCGame.play (/mnt/dataset/unchecked_low_level_calls/0x07f7ecb66d788ab01dc93b9b71a88401de7d0f2e.sol#147-179) uses a dangerou
              },
              "name": "OtherVulnerability",
              "shortDescription": {
                "text": "incorrect-equality"
              }
            },
            {
              "id": "Reentrancy_9",
              "fullDescription": {
                "text": "Reentrancy in PoCGame.donateToWhale (/mnt/dataset/unchecked_low_level_calls/0x07f7ecb66d788ab01dc93b9b71a88401de7d0f2e.sol#2
              },
              "name": "ReentrancyVulnerability",
              "shortDescription": {
```

Figure 4.1: Upload of a correctly formatted SARIF file on the Microsoft's Validation tool (From: [5])



SARIF Validator

Validation complete!

JSON1020: An array's elements are not unique, as required by the schema's "uniqueItems" property.

```
                  "snippet": {},
                  "startColumn": 10,
                  "startLine": 20
                }
              }
            }
          ],
          "ruleId": "FrontRunning_6"
        }
      ]
    },
    {
      "tool": {
        "driver": {
          "name": "Securify",
          "fullDescription": {
            "text": "Securify uses formal verification, also relying on static analysis checks. Securify\u2019s analysis consists of two steps. Fi
          },
          "informationUri": "https://github.com/eth-sri/securify2",
          "rules": [
            {
              "id": "FrontRunning_3",
              "name": "FrontRunningVulnerability",
              "shortDescription": {
                "text": "TODReceiver"
              }
            },
            {
              "id": "UncheckedLowCalls_4",
              "name": "UncheckedLowCallsVulnerability",
              "shortDescription": {
                "text": "UnhandledException"
              }
            },
            {
              "id": "FrontRunning_4",
              "name": "FrontRunningVulnerability",
```

JSON1020: at "runs[4].tool.driver.rules": The elements of the array are not unique.

Figure 4.2: Upload of a badly formatted SARIF file on the Microsoft's Validation tool (From: [5])

This tool comes with a feature that checks as well for the GitHub ingestion rules. This option saved us a lot of work in Section 4.3 as it is no longer required for us to upload all of $SB^{CURATED}$ dataset to assert the output's correctness. This verification serves the purpose of acknowledging that the entire dataset's results are correctly formatted for display on the SARIF processing mechanism from GitHub.

4.1.3 Vulnerabilities Verification

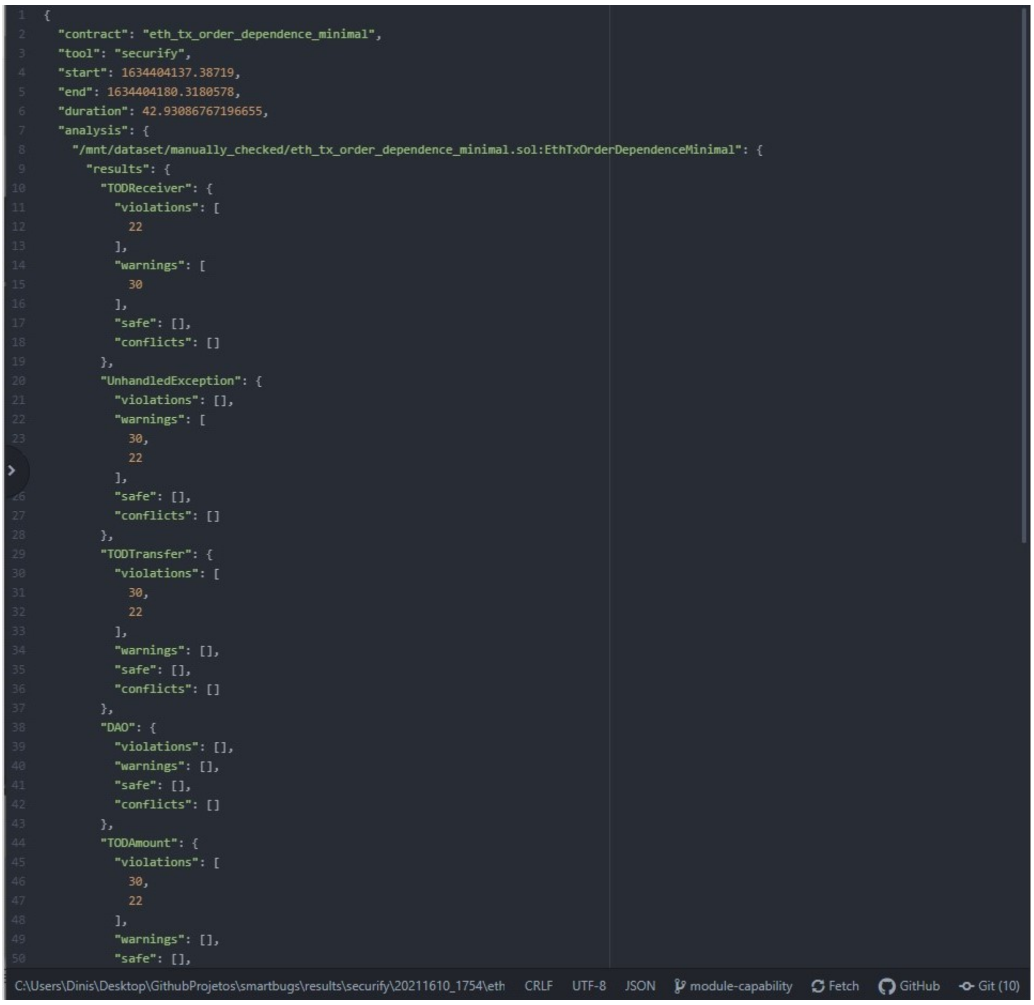
From the $SB^{CURATED}$ dataset we retrieved an arbitrary subset consisting of one random smart contract per DASP-10 category. We chose one from each category so that we could manually verify our converters against the most diversified outputs. The final subset is comprised by the following contracts:

- mycontract.sol - Access Control
- integer_overflow_1.sol - Arithmetic
- guess_the_random_number.sol - Bad Randomness
- dos_simple.sol - Denial of Service
- eth_tx_order_dependence_minimal.sol - Front Running
- name_registrar.sol - Other
- simple_dao.sol - Reentrancy
- short_address_example.sol - Short Addresses
- roulette.sol - Time Manipulation
- lotto.sol - Unchecked Low Level Calls

These 10 smart contracts were analyzed by the 11 tools producing 110 result files. The results were manually checked for the integrity of the translation from the original output to SARIF. As expected, every correctly formatted vulnerability was successfully translated. While most of the vulnerabilities that were not correctly formatted was due to lack of line information, we also noticed that there were some that reported a vulnerability at line ”-1” and a few ones that reported something without any context or information. We marked these cases as correctly discarded.

Even though SARIF is comprehensive enough to be the world's standard for all static analyzers, unfortunately it lacks some simplicity and intuitiveness when a user tries to read directly from the SARIF file. So in order to ease the manual comparison of the original output with the SARIF and to reduce the possibility of human mistakes, we took the liberty of using another free Microsoft provided tool for SARIF files.

The SARIF Web Viewer[2] is a tool that allows any developer to see a robust interface detailing the SARIF file uploaded by the user. We recommend this interface for testing and viewing SARIF files in a simplified manner. An example of the comparison between the original output represented by the text editor Atom[3] and the SARIF Viewer can be seen in Figure 4.3 and 4.4. The output represented in the Figure corresponds to the analysis of the eth_tx_order_dependence_minimal.sol performed by Securify.

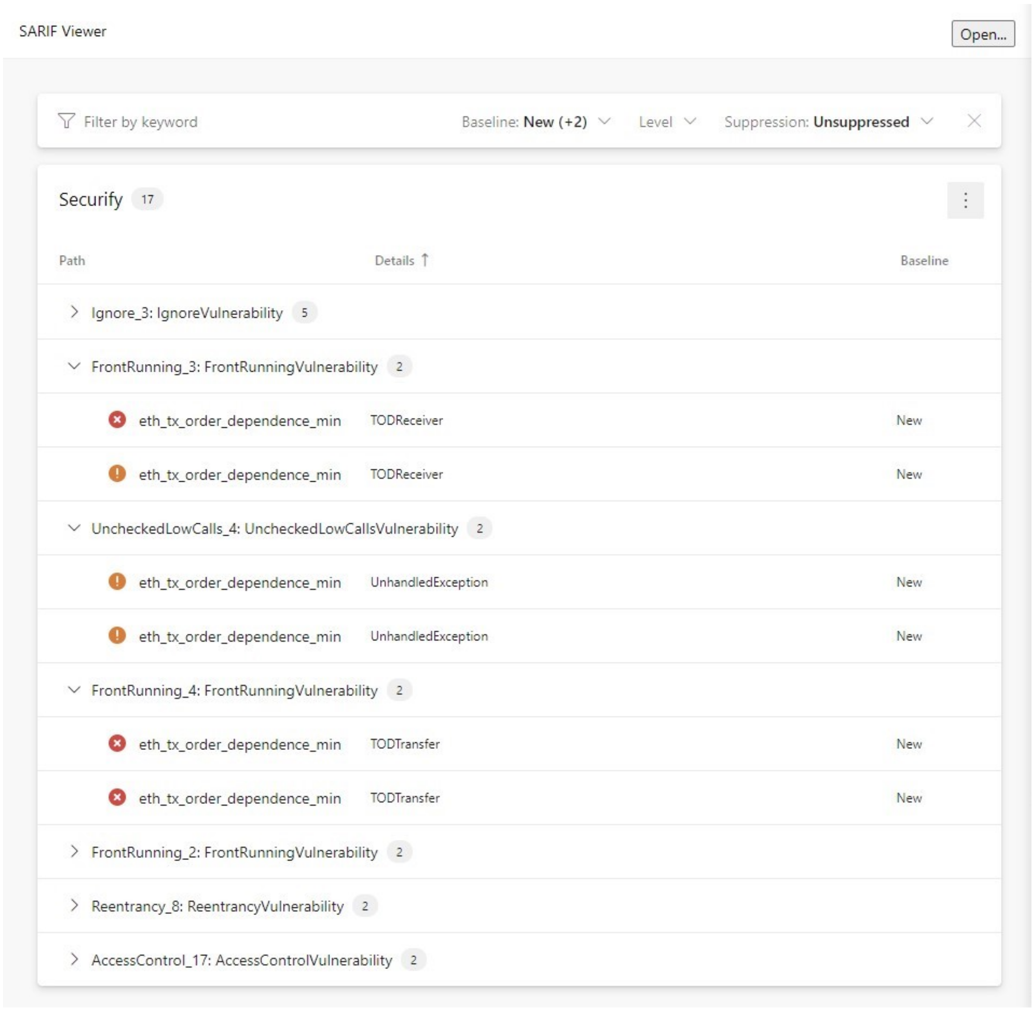


```
{
  "contract": "eth_tx_order_dependence_minimal",
  "tool": "securify",
  "start": 1634404137.38719,
  "end": 1634404180.3180578,
  "duration": 42.93086767196655,
  "analysis": {
    "/mnt/dataset/manually_checked/eth_tx_order_dependence_minimal.sol:EthTxOrderDependenceMinimal": {
      "results": {
        "TODReceiver": {
          "violations": [
            22
          ],
          "warnings": [
            30
          ],
          "safe": [],
          "conflicts": []
        },
        "UnhandledException": {
          "violations": [],
          "warnings": [
            30,
            22
          ],
          "safe": [],
          "conflicts": []
        },
        "TODTransfer": {
          "violations": [
            30,
            22
          ],
          "warnings": [],
          "safe": [],
          "conflicts": []
        },
        "DAO": {
          "violations": [],
          "warnings": [],
          "safe": [],
          "conflicts": []
        },
        "TODAmount": {
          "violations": [
            30,
            22
          ],
          "warnings": [],
          "safe": [],
```

C:\Users\Dinis\Desktop\GithubProjetos\smartbugs\results\securify\20211610_1754\eth
CRLF
UTF-8
JSON
module-capability
Fetch
GitHub
Git (10)

Figure 4.3: Original output of the eth_tx_order_dependence_minimal.sol analysis by Securify

[2]SARIF Web Viewer: https://microsoft.github.io/sarif-web-component/
[3]Atom: https://atom.io/



Path	Details	Baseline
Ignore_3: IgnoreVulnerability 5		
FrontRunning_3: FrontRunningVulnerability 2		
eth_tx_order_dependence_min	TODReceiver	New
eth_tx_order_dependence_min	TODReceiver	New
UncheckedLowCalls_4: UncheckedLowCallsVulnerability 2		
eth_tx_order_dependence_min	UnhandledException	New
eth_tx_order_dependence_min	UnhandledException	New
FrontRunning_4: FrontRunningVulnerability 2		
eth_tx_order_dependence_min	TODTransfer	New
eth_tx_order_dependence_min	TODTransfer	New
FrontRunning_2: FrontRunningVulnerability 2		
Reentrancy_8: ReentrancyVulnerability 2		
AccessControl_17: AccessControlVulnerability 2		

Figure 4.4: SARIF output of the eth_tx_order_dependence_minimal.sol analysis by Securify processed by Microsoft's SARIF Viewer tool (From: [6])

4.2 SASP Service

In this stage, we tested our server implementation by querying a smaller set of the vulnerable smart contracts. It is expected from the SASP service an output SARIF file including all security standards for the request and the response produced by the server. In the case where the query does not comply by them, SASP should reject and ignore the request. Furthermore, we will stress test our server so that we may acknowledge the amount of parallel executions our service can sustain.

4.2.1 Functionality

Firstly we tested our server's primary functionality, to be able to receive a request from a user, run the analysis and reply with a single comprehensive SARIF formatted file representing all of the analysis information. After reading the response file on the client side and compare it against a local report we reached the conclusion that so far everything was working properly.

Moreover we went and performed the same test on the GitHub machine to affirm the correctness of running on different machines. After ordering the analysis we printed the file to the console and confirmed it was in fact the correct report with no information lost.

Secondly, we sent multiple badly formed requests with the following characteristics:

- A badly formed user-hash,
- No user-hash,
- Non existing tool selected,
- No tool selected,
- With a directory transversal exploit both in the user-hash and in the files names.

The SASP service behaved exactly as expected, ignoring the requests when it was required. In the cases where it was deemed feasible altering the query to comply with the criteria, the service also behaved correctly. More specifically, the case with the exploit for the directory transversal attack was correctly mitigated by the algorithm provided by [60].

4.2.2 Stress Test

Finally, we decided to conduct a stress test to see how our server would behave when queried by multiple requests from different users. All of the tests were performed under the worst-case scenario where the server is the most overloaded. We created this scenario by guaranteeing that the analysis requests were sent all at the same time. With this test we hope to deduce the approximate scalability of our developed solution with the purpose of preventing clogged or crashed services.

For this part of our evaluation we developed a script that requests multiple analysis on different threads at the same time. We performed two stress tests. One for low time consuming analysis where we employed Oyente analysing the SimpleDAO contract. And another for a more CPU intense analysis combining Mythril and Slither (due to their best combined performance) analyzing two contracts, the SimpleDAO and the Reentrance. To account for errors in measuring time we ran the script three times and calculated the median. We executed the tests from 1 to 45 simultaneous clients. The time measured comprises all sent requests, the analysis execution and the results download for each client. The Oyente Graph can be seen in Figure 4.5 and the high CPU consuming Graph in Figure 4.6.

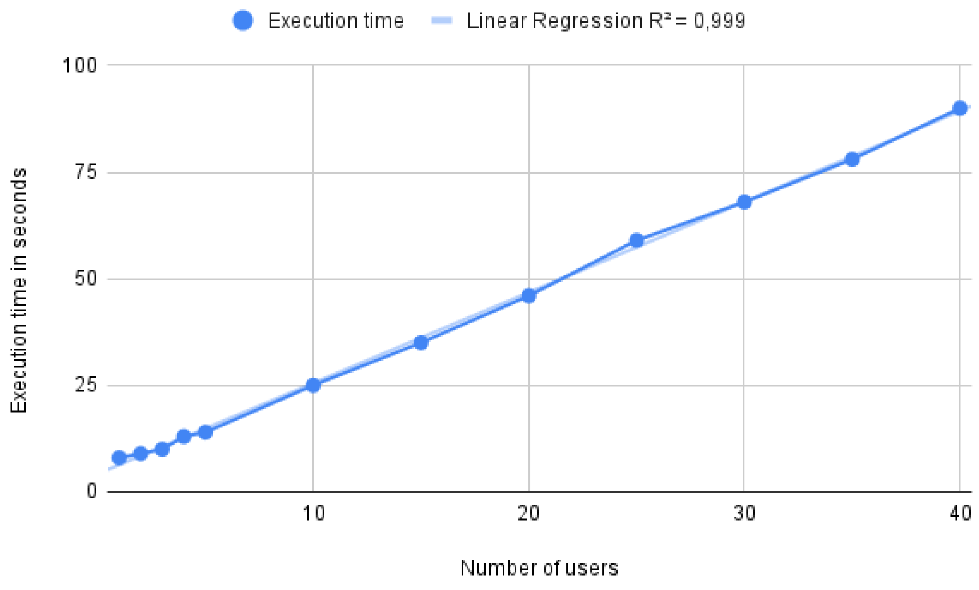


Figure 4.5: SASP stress testing with Oyente analysing SimpleDAO

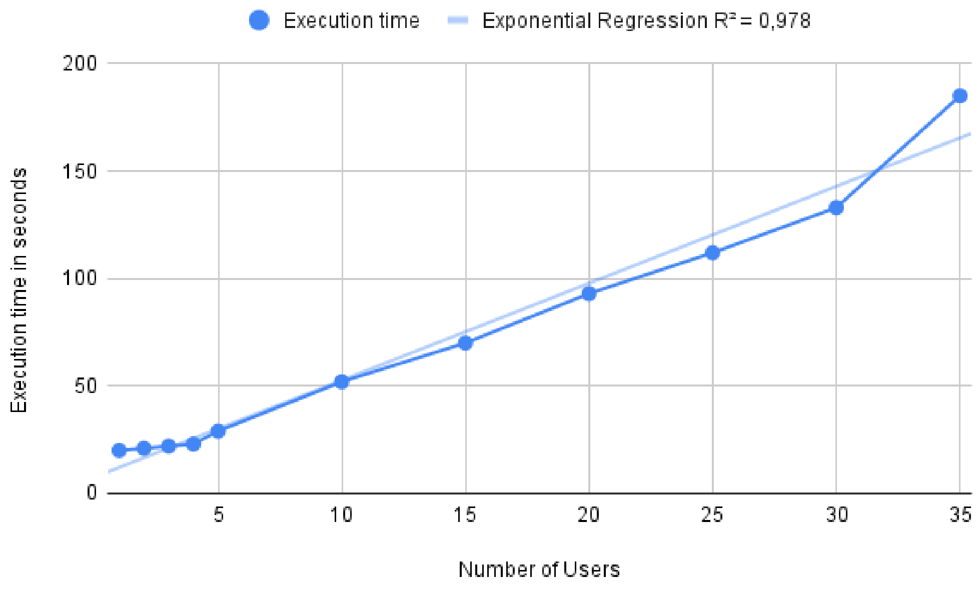


Figure 4.6: SASP stress testing with Mythril and Slither analysing SimpleDAO and Reentrancy

For our surprise, the SASP service is more scalable than we had anticipated. As we can see in the Oyente graph, there is a linear regression between the execution time and the number of users with a R^2 of 0.99. This variable being this close to 1 means that the linear approximation of the results is a good fit and therefore we can conclude that the time increases constantly as per number of clients requesting our service.

On the other hand, as we can see in the Mythril and Slither graph, when performing a more intense analysis, the services starts to lose performance at 35 simultaneous requests.

At 45 and 40 simultaneous analyses, when analyzing with Oyente and with Mythril and Slither correspondingly, Docker service starts to crash and consequently SASP stops all current analysis so that it may recover from the crash. So, we conclude that the current bottleneck for our platform lies in the Docker Python API service.

Since it is Docker itself that controls its threads' priority and execution, and that it allows for an arbitrary number of concurrent containers, this error should not happen and therefore we are unable to take further conclusions about our system. As such there is an existing issue on GitHub that is being assessed, contemplating this Docker bug [61].

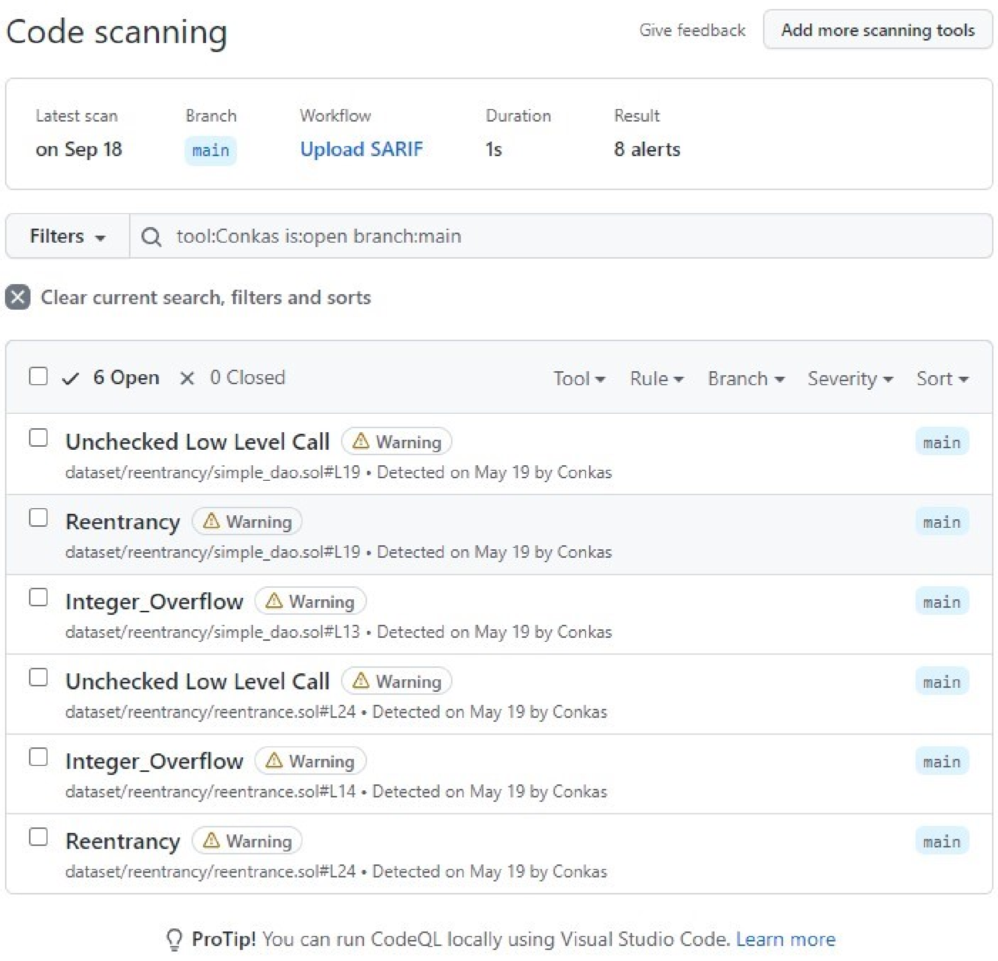
4.3 GitHub Adapter

After completing the previous stages, we evaluated our GitHub adapter. In this stage we will need to upload a subset of the $SB^{CURATED}$ to a GitHub repository and verify 3 distinct parts: the functionality, the uniqueness and the intuitiveness.

4.3.1 Functionality

For the functionality part, we will establish that all smart contracts uploaded are being correctly queried both to our SASP and NoSASP service and that the adapter is accurately processing the response, by comparing the outcome with the output produced by SARIF's converter.

As we tested the entire $SB^{CURATED}$ on the SARIF Validation Tool with the GitHub ingestion feature that we have seen in Section 4.1.2, there was no need to manually verify the entire dataset. Therefore we chose two contracts to manually ensure this stage of the evaluation. After uploading the small subset to GitHub and analyzing it with the 11 available tools, we were able to verify that everything is working correctly. A direct comparison from the GitHub vulnerability report with the file produced by Conkas' SARIF converter can be seen in Figure 4.7 and 4.8.



Code scanning

Give feedback | Add more scanning tools

Latest scan	Branch	Workflow	Duration	Result
on Sep 18	main	Upload SARIF	1s	8 alerts

Filters ▾ | tool:Conkas is:open branch:main

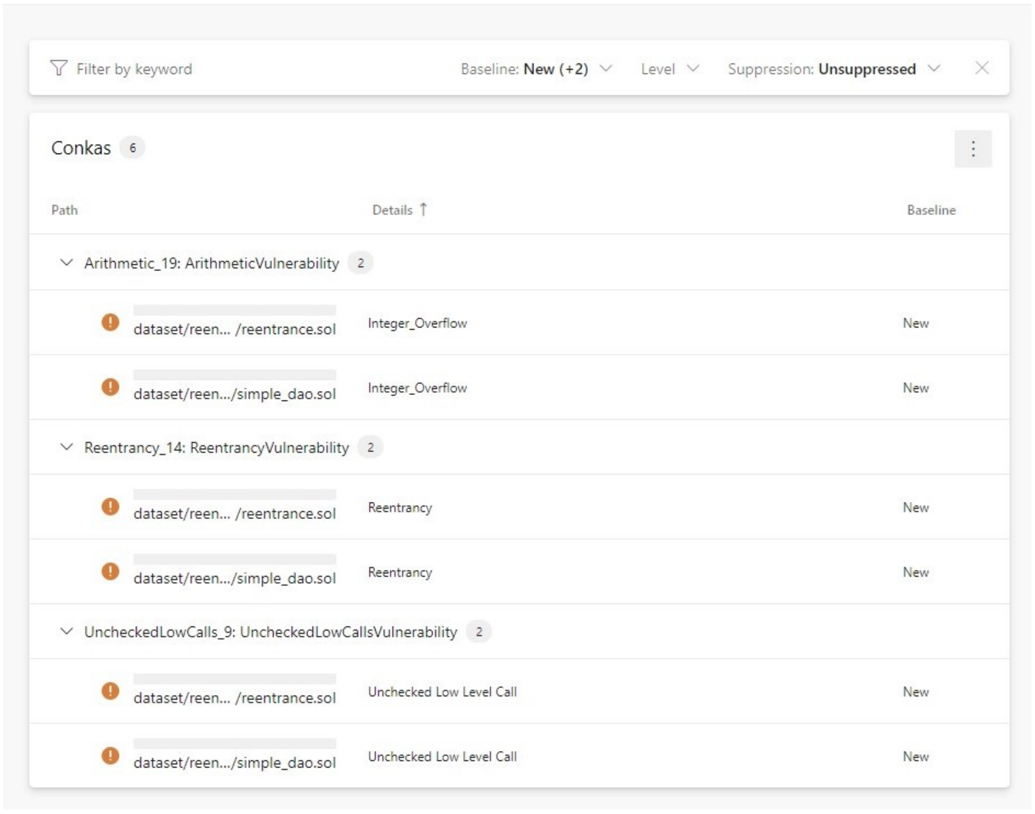
Clear current search, filters and sorts

6 Open ✕ 0 Closed — Tool ▾ Rule ▾ Branch ▾ Severity ▾ Sort ▾

Alert	Severity	Location	Branch
Unchecked Low Level Call	⚠ Warning	dataset/reentrancy/simple_dao.sol#L19 • Detected on May 19 by Conkas	main
Reentrancy	⚠ Warning	dataset/reentrancy/simple_dao.sol#L19 • Detected on May 19 by Conkas	main
Integer_Overflow	⚠ Warning	dataset/reentrancy/simple_dao.sol#L13 • Detected on May 19 by Conkas	main
Unchecked Low Level Call	⚠ Warning	dataset/reentrancy/reentrance.sol#L24 • Detected on May 19 by Conkas	main
Integer_Overflow	⚠ Warning	dataset/reentrancy/reentrance.sol#L14 • Detected on May 19 by Conkas	main
Reentrancy	⚠ Warning	dataset/reentrancy/reentrance.sol#L24 • Detected on May 19 by Conkas	main

ProTip! You can run CodeQL locally using Visual Studio Code. Learn more

Figure 4.7: GitHub SARIF interface after consuming the output sent by the SASP service.



Conkas 6

Path	Details	Baseline
Arithmetic_19: ArithmeticVulnerability 2		
dataset/reen... /reentrance.sol	Integer_Overflow	New
dataset/reen.../simple_dao.sol	Integer_Overflow	New
Reentrancy_14: ReentrancyVulnerability 2		
dataset/reen... /reentrance.sol	Reentrancy	New
dataset/reen.../simple_dao.sol	Reentrancy	New
UncheckedLowCalls_9: UncheckedLowCallsVulnerability 2		
dataset/reen... /reentrance.sol	Unchecked Low Level Call	New
dataset/reen.../simple_dao.sol	Unchecked Low Level Call	New

Figure 4.8: SARIF Viewer displaying the SARIF file produced by SmartBugs locally.

4.3.2 Uniqueness

For the uniqueness feature, we had already verified that when consuming a duplicate vulnerability, GitHub's SARIF processing mechanism automatically marks it as rediscovered instead of detecting a new and different one. Moreover, a developer is able to tag each vulnerability with multiple labels, as we have seen in Section 3.4.2, so as to not be warned over a unwanted vulnerability report. Without the uniqueness of the reported vulnerabilities our service could result in confusing feedback for the smart contract developer.

4.3.3 Intuitiveness

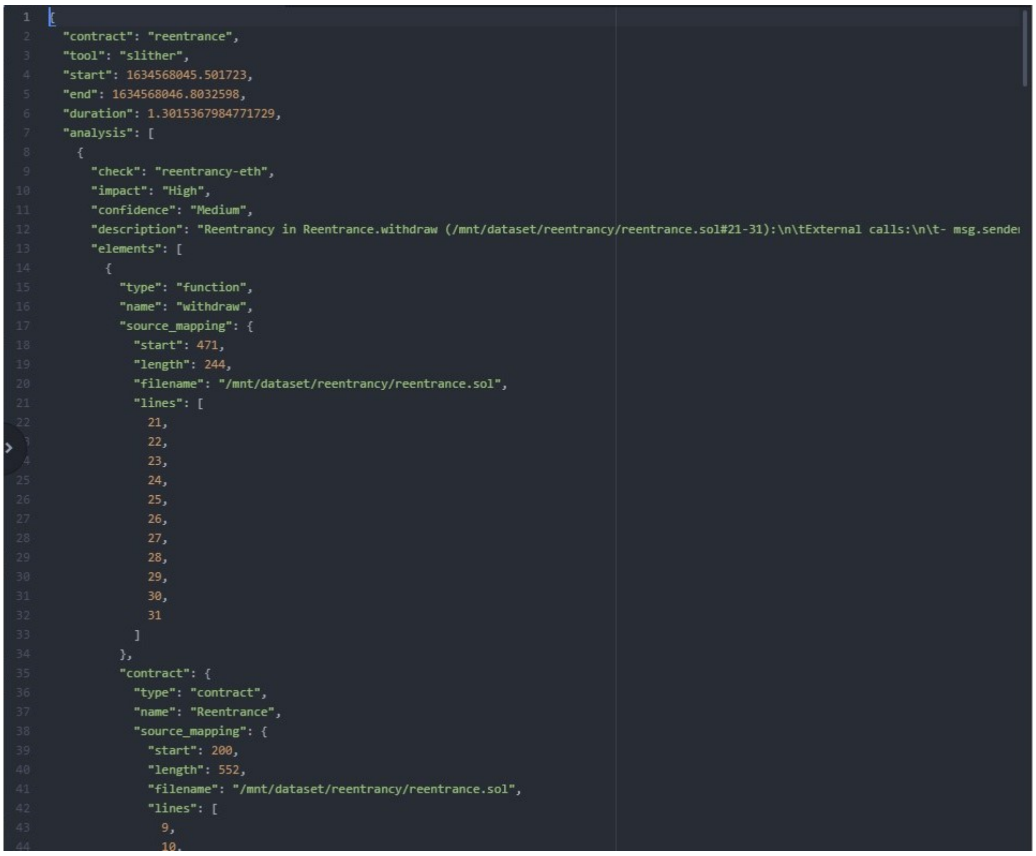
At last, we will discuss if the comments and the system as a whole are intuitive enough for any programmer to be able to abstract from the analysis and only focus on coding the smart contract itself. This was the trickiest part of our evaluation since not every developer will agree on this. It can be either from the false positives problem, the execution time of the analysis or even due to the discordance between the vulnerabilities reported by the tools.

Nevertheless, we believe that in comparison to the previous implemented system, we managed to increase the intuitiveness of SmartBugs from multiple perspectives.

First of all we have increased the easiness and lowered the human input necessary to run an analysis. Before a developer had to download SmartBugs, Python and Docker, install all required modules and manually start the analysis every time there was a need to do so. Now there is only the need to add a workflow when running the analysis for the first time and specify when GitHub is to repeat it.

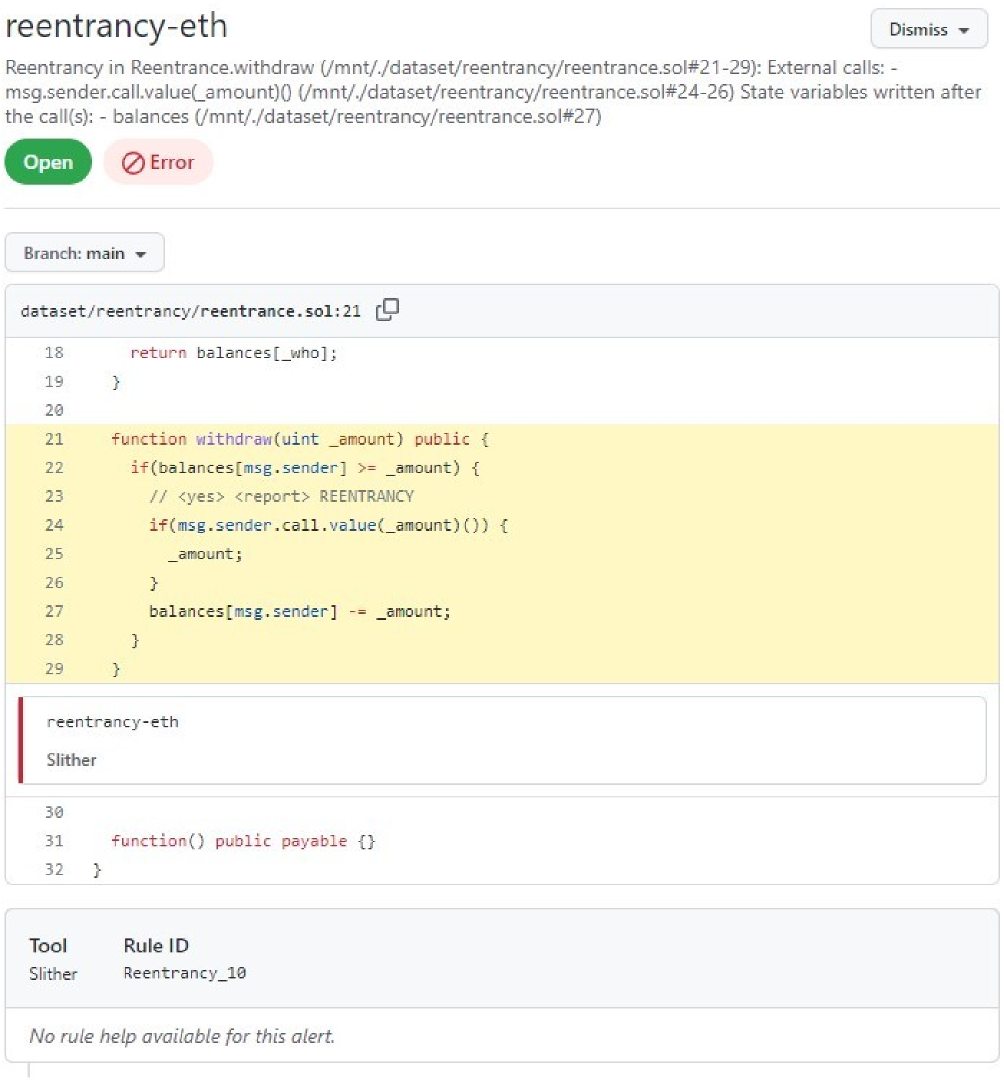
Secondly we substantially reduced the number of outputs that a developer is required to read. For example, if a programmer wanted to analyze 2 smart contracts by 3 tools, he would be compelled to open and acknowledge 6 output files, on 3 distinct formats. With SmartBugs 2.0 the number of outputs was significantly reduced from the number of tools times the number of smart contracts to one single output file.

Finally, we feel that displaying the results in a coding interface is an intuitiveness improvement compared to the previous system. Not only for the simplicity in acknowledging the vulnerability but also because it becomes easier to discard and ignore unwanted ones. An example of the before and after of the Slither output can be seen in Figure 4.9 and 4.10.



```
{
  "contract": "reentrance",
  "tool": "slither",
  "start": 1634568045.501723,
  "end": 1634568046.8032598,
  "duration": 1.3015367984771729,
  "analysis": [
    {
      "check": "reentrancy-eth",
      "impact": "High",
      "confidence": "Medium",
      "description": "Reentrancy in Reentrance.withdraw (/mnt/dataset/reentrancy/reentrance.sol#21-31):\n\tExternal calls:\n\t- msg.sende
      "elements": [
        {
          "type": "function",
          "name": "withdraw",
          "source_mapping": {
            "start": 471,
            "length": 244,
            "filename": "/mnt/dataset/reentrancy/reentrance.sol",
            "lines": [
              21,
              22,
              23,
              24,
              25,
              26,
              27,
              28,
              29,
              30,
              31
            ]
          },
          "contract": {
            "type": "contract",
            "name": "Reentrance",
            "source_mapping": {
              "start": 200,
              "length": 552,
              "filename": "/mnt/dataset/reentrancy/reentrance.sol",
              "lines": [
                9,
                10,
```

Figure 4.9: The original Slither output.


reentrancy-eth

Reentrancy in Reentrance.withdraw (/mnt/./dataset/reentrancy/reentrance.sol#21-29): External calls: - msg.sender.call.value(_amount)() (/mnt/./dataset/reentrancy/reentrance.sol#24-26) State variables written after the call(s): - balances (/mnt/./dataset/reentrancy/reentrance.sol#27)

dataset/reentrancy/reentrance.sol:21

```
      return balances[_who];
    }

    function withdraw(uint _amount) public {
      if(balances[msg.sender] >= _amount) {
        // <yes> <report> REENTRANCY
        if(msg.sender.call.value(_amount)()) {
          _amount;
        }
        balances[msg.sender] -= _amount;
      }
    }
```

reentrancy-eth

Slither

```

    function() public payable {}
  }
```

Tool	Rule ID
Slither	Reentrancy_10

No rule help available for this alert.

Figure 4.10: GitHub SARIF interface after consuming the output produced by Slither.

5 Conclusion and Future Work

Contents

5.1 Achievements 71
5.2 Future Work 72

In this chapter, we discuss the dissertation's significant contributions while evaluating whether our goals were realized. We also talk about future developments on how to improve our work.

5.1 Achievements

As smart contracts popularity, use and financial value increases, more vulnerabilities with bigger repercussions are discovered. There is still not only a lot of research lacking in this field but also some scarcity of awareness towards the issues these vulnerabilities represent.

As the absence of executing static analysis tools is often related to the difficulty of running the analysis and the complexity of acknowledging its output, in this thesis we describe a method to expand not just SmartBugs but any static analysis tool created by making it more available and accessible to a developer. We discuss standardized methodologies to ease the use of static analysis and furthermore we describe a method to inflate the code with the analysis' results so that a developer may abstract himself of the analysis and treat a vulnerability just like an minor error message.

After our literature review we describe our implementations into a developer's environment. We started by organizing the SARIF standard in a convenient and accessible manner, paving the way for adoption from other static analysis tools. Following this, we expose our vision and implementation for the SASP protocol. We also describe both GitHub adapters developed. One relying on the SASP server and another independent from this service in spite of the added limitations. Finally we thoroughly explain the changes and new features for SmartBugs itself.

Using SmartBugs, we were able to verify the correctness of our system. Firstly we fixed our converters from each tool's original output to SARIF based on the validity of the output produced when analyzing the entire $SB^{CURATED}$ dataset. Furthermore we manually checked a smaller subset of outputs to increase our converters' trustworthiness. We also tested SASP both on the functionality and on the amount of work it is able to sustain, leading to a maximum of 35 users concurrently analyzing with our service. At last, we tested both GitHub adapters for their functionality, uniqueness and intuitiveness where we found that everything was working as expected.

We achieved the goals we set for ourselves through our work. A visual representation of our work and the expanded new possibilities for a Solidity developer was shown in Figure 3.1. For the future we would like to see other static analysis tools playing SmartBugs' role in this diagram.

5.2 Future Work

Static analysis research has received significant contributions in recent years, specially for smart contracts as mentioned throughout this article. Because Blockchain and smart contracts are evolving at such a rapid speed, we should strive to keep our work up to date. The following are some possibilities for future work:

- **Distributed SASP** - Depending on the SASP adoption, it can be deployed on a distributed system in order to be able to withstand a bigger workload. At the current moment we believe the optimal maximum number of users should be between 35 to 40, however with a distributed system this number could be increased.
- **Update SmartBugs and its tools** - SmartBugs should be improved, and additional tools could be introduced. The current 11 tools only work on some versions of Solidity which can render some of them useless specially when analyzing more recent Solidity versions. Not just SmartBugs but also the tools themselves should be kept updated by their developers or the community.
- **Improve $SB^{CURATED}$ and SARIF fields** - The $SB^{CURATED}$ dataset can be expanded, and it should be updated from time to time to include additional contracts and possibly support new sorts of vulnerabilities. As well as more SARIF keys should be added to improve readability from the user when running an analysis.
- **More adoption for SARIF** - More and more static analysis tools keep on being created in their own distinct format, which leads to a decreased usage on their clients part. Developers could even employ our work to ease the implementation of SARIF into their tools' environment. SARIF standardizes not only Ethereum smart contracts' analyzers but every static analyzer and it even works on some dynamic ones, so there's no bounding limit to where it can be implemented.

Bibliography

[1] S. Nakamoto. (2008) Bitcoin: A peer-to-peer electronic cash system. [Online]. Available: www.bitcoin.org/bitcoin.pdf

[2] A. P. C. Monteiro, “A study of static analysis tools for ethereum smart contracts,” 2019.

[3] P. Anderson, “Modernizing static analysis tools to facilitate integrations,” *ACM SIGAda Ada Letters*, vol. 39, no. 1, pp. 101–108, 2020.

[4] Z. Peng and X. Ma, “Exploring how software developers work with mention bot in github,” *CCF Transactions on Pervasive Computing and Interaction*, p. 190, 2019. [Online]. Available: https://doi.org/10.1007/s42486-019-00013-2

[5] Microsoft. (2020) Sarif validator. [Online]. Available: https://sarifweb.azurewebsites.net/Validation

[6] M. Corporation. (2020) Sarif viewer. [Online]. Available: https://microsoft.github.io/sarif-web-component/

[7] N. Veloso. (2021) nveloso/conkas: Ethereum virtual machine (evm) bytecode or solidity smart contract static analysis tool based on symbolic execution. [Online]. Available: https://github.com/nveloso/conkas#readme

[8] A. Rosic. What is blockchain technology? a step-by-step guide for beginners. [Online]. Available: https://blockgeeks.com/guides/what-is-blockchain-technology/

[9] A. M. Antonopoulos, *Mastering Bitcoin: unlocking digital cryptocurrencies*. ” O’Reilly Media, Inc.”, 2014.

[10] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck, “Blockchain,” *Business & Information Systems Engineering*, vol. 59, no. 3, pp. 183–187, 2017.

[11] V. Buterin, “Ethereum whitepaper,” 2013. [Online]. Available: https://ethereum.org/en/whitepaper/

[12] N. Szabo, “Formalizing and securing relationships on public networks,” *First Monday*, 1997.

[13] A. M. Antonopoulos and G. Wood, *Mastering ethereum: building smart contracts and dapps*. O'reilly Media, 2018.

[14] L. Hollander. (2019) The ethereum virtual machine — how does it work? [Online]. Available: https://medium.com/mycrypto/the-ethereum-virtual-machine-how-does-it-work-9abac2b7c9e

[15] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 653–663.

[16] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 254–269.

[17] Z. A. Khan and A. S. Namin, "A survey on vulnerabilities of ethereum smart contracts," *arXiv preprint arXiv:2012.14481*, 2020.

[18] B. C. Gupta, N. Kumar, A. Handa, and S. K. Shukla, "An insecurity study of ethereum smart contracts," in *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 2020, pp. 188–207.

[19] R. O. Team. (2020) Cover infinite mint exploit. [Online]. Available: https://www.notion.so/Cover-Infinite-Mint-Exploit-0a234cc279484982ae559bb5ab54532a

[20] K. J. Kistner. (2020) itoken duplication incident report. [Online]. Available: https://bzx.network/blog/incident

[21] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: a framework to analyze solidity smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1349–1352.

[22] Y. Ding, C. Wang, Q. Zhong, H. Li, J. Tan, and J. Li, "Function-level dynamic monitoring and analysis system for smart contract," *IEEE Access*, 2020.

[23] M. Romiti, A. Judmayer, A. Zamyatin, and B. Haslhofer, "A deep dive into bitcoin mining pools: An empirical analysis of mining shares," *arXiv preprint arXiv:1905.05999*, 2019.

[24] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[25] N. Szabo, "Smart contracts: building blocks for digital markets," *EXTROPY: The Journal of Transhumanist Thought,(16)*, vol. 18, no. 2, 1996.

[26] Solidity. Solidity documentation. [Online]. Available: https://docs.soliditylang.org/en/v0.7.5/

[27] N. Gruhn. (2019) What makes a programming language turing complete? [Online]. Available: https://dev.to/gruhn/what-makes-a-programming-language-turing-complete-58fl

[28] R. Fontein, "Comparison of static analysis tooling for smart contracts on the evm," in *28th Twente Student conference on IT*, 2018.

[29] NCCGroup. (2018) Decentralized application security project (or dasp) top 10. [Online]. Available: https://dasp.co/

[30] P. Daian. (2016) Analysis of the dao exploit. [Online]. Available: https://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[31] T. Gagliardoni. (2021) The poly network hack explained – kudelski security research. [Online]. Available: https://research.kudelskisecurity.com/2021/08/12/the-poly-network-hack-explained/

[32] M. Qi, Z. Wang, D. Liu, Y. Xiang, B. Huang, and F. Zhou, "Acctp: Cross chain transaction platform for high-value assets," in *International Conference on Blockchain*. Springer, 2020, pp. 154–168.

[33] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[34] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[35] I. Grishchenko, M. Maffei, and C. Schneidewind, "A semantic framework for the security analysis of ethereum smart contracts," in *International Conference on Principles of Security and Trust*. Springer, 2018, pp. 243–269.

[36] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts." in *Ndss*, 2018, pp. 1–12.

[37] D. Perez and B. Livshits, "Smart contract vulnerabilities: Does anyone care?" *arXiv preprint arXiv:1902.06710*, pp. 1–15, 2019.

[38] M. Di Angelo and G. Salzer, "A survey of tools for analyzing ethereum smart contracts," in *2019 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPCON)*. IEEE, 2019, pp. 69–78.

[39] C. F. Torres, M. Steichen *et al.*, “The art of the scam: Demystifying honeypots in ethereum smart contracts,” in *28th {USENIX} security symposium ({USENIX} security 19)*, 2019, pp. 1591–1607.

[40] J. Iadeluca, “What is maian: The most popular smart contract analysis tool,” *https://btcmanager.com/*, 2019. [Online]. Available: https://btcmanager.com/maian-smart-contract-analysis-tool/

[41] P. Szilágyi, “55 percent against ethereum’s parity wallet from returning “locked-up” ether — btcmanager,” *https://btcmanager.com/*, 2018. [Online]. Available: https://btcmanager.com/55-percent-against-ethereums-parity-wallet-from-returning-locked-up-ether/

[42] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, “Manticore: A user-friendly symbolic execution framework for binaries and smart contracts,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

[43] B. Mueller, “Smashing ethereum smart contracts for fun and real profit,” in *9th Annual HITB Security Conference (HITBSecConf)*, vol. 54, 2018.

[44] 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*, 2018, pp. 664–676.

[45] 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)*. IEEE, 2019, pp. 8–15.

[46] P. Fire. (2017) Solhint: An advanced linter for ethereum’s solidity. [Online]. Available: https://medium.com/protofire-blog/solhint-an-advanced-linter-for-ethereums-solidity-c6b155aced7b

[47] 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*, 2020, pp. 530–541.

[48] R. M. Parizi, A. Dehghantanha, K.-K. R. Choo, and A. Singh, “Empirical vulnerability analysis of automated smart contracts security testing on blockchains,” *arXiv preprint arXiv:1809.02702*, 2018.

[49] P. Anderson. (2018) Static analysis results: A format and a protocol: Sarif & sasp. [Online]. Available: https://blogs.grammatech.com/static-analysis-results-a-format-and-a-protocol-sarif-sasp

[50] P. Anderson, Ł. Kot, N. Gilmore, and D. Vitek, “Sarif-enabled tooling to encourage gradual technical debt reduction,” in *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE, 2019, pp. 71–72.

[51] L. Luo, J. Dolby, and E. Bodden, "Magpiebridge: A general approach to integrating static analyses into ides and editors (tool insights paper)," in *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[52] Oasis. Static analysis results interchange format (sarif) version 2.1.0. [Online]. Available: docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html

[53] S. Kummita and G. Piskachev, "Integration of the static analysis results interchange format in cognicrypt," *arXiv preprint arXiv:1907.02558*, 2019.

[54] ISO. Data elements and interchange formats — information interchange — representation of dates and times. [Online]. Available: https://www.iso.org/standard/40874.html

[55] Microsoft. (2021) microsoft/sarif-tutorials: User-friendly documentation for the sarif file format. [Online]. Available: https://github.com/microsoft/sarif-tutorials

[56] GitHub. Uploading a sarif file to github. [Online]. Available: https://docs.github.com/en/free-pro-team@latest/github/finding-security-vulnerabilities-and-errors-in-your-code/uploading-a-sarif-file-to-github

[57] M. Fowler. (2006) Continuous integration. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[58] B. Vasilescu, S. Van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. van den Brand, "Continuous integration in a social-coding world: Empirical evidence from github," in *2014 IEEE international conference on software maintenance and evolution*. IEEE, 2014, pp. 401–405.

[59] N. Cassee, B. Vasilescu, and A. Serebrenik, "The silent helper: the impact of continuous integration on code reviews," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 423–434.

[60] M. Flanders, "A simple and intuitive algorithm for preventing directory traversal attacks," *arXiv preprint arXiv:1908.04502*, 2019.

[61] sheridp. (2018) Container.wait with timeout raises connection error · issue 1966 · docker/docker-py. [Online]. Available: https://github.com/docker/docker-py/issues/1966