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

Dinis Antunes Palha de Araújo
dinisaraujo@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2021

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; SARIF; SASP; GitHub

1. Introduction

Blockchain technology [28] has gained massive attention since the introduction of Bitcoin's white paper in 2008 [22]. 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. In this network no node trusts each other, however due to its cryptographic properties, they will always agree by maintaining a shared global ledger while overcoming the missing trust [3, 25].

Since Bitcoin's introduction, several other Blockchain platforms, extending the technology's applications, have emerged. In particular, Ethereum [5, 4] 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 [32] into the Blockchain technology.

Due to the rich 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 [24]. In another study performed on 19,366 existing Ethereum contracts using Oyente[2], 8,833 were flagged as vulnerable, including the TheDAO vulnerability which led to a 60 million US dollars loss in June 2016[20].

In order to detect bugs and security flaws in their Smart Contracts' code [16, 13], developers use static analysis tools to analyse their code. These are ultimately necessary for software developers to find simple errors, serious vulnerabilities, performance issues, libraries' misuses[3], and even to standout de-

[1]`https://github.com/ivicanikolicsg/MAIAN`

[2]`https://github.com/enzymefinance/oyente`

[3]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

sign flaws[4].

According to a set of criteria [21], 10 static analysis tools were selected to be a part of SmartBugs [9]. SmartBugs was created with the purpose of being an extensible execution framework that simplifies the execution of analysis tools on smart contracts.

Many static analysis tools have been created in academia but only a few are actively used by developers [7]. 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. 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.

1.1. Objectives and Contributions

Our major goal in this project 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 will 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 SARIF standard. We present our overview on this standard and provide a minimalist way to implement it into any static analyzer's environment, for Solidity smart 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 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.

Our major contributions are enumerated here:

- We briefly review the vulnerabilities found in smart contracts and a state-of-the-art tool, SmartBugs, for the automated identification of these vulnerabilities.
- We present an overview on SARIF, a standard for static analysis tools' output. We implement it into the SmartBugs' environment[5] and provide a more accessible way to implement it in other tools.
- We develop SASP[6] 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[7] for requesting static analysis results from SASP and directly from SmartBugs.
- Lastly, we test all our implementations from different perspectives in order to provide a fair evaluation.

2. Background

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.

Thanks to Satoshi Nakamoto, the pseudonym behind the Bitcoin paper, for proposing the first permissionless blockchain in 2008, the Bitcoin white paper [22]. As it's a permissioneless blockchain, any user is able to interact and participate in the network by simply publishing a signed transaction. Satoshi not only published the white paper as well

developers allowed for a serious security vulnerability in the *updatePool* function to be exploited [33].

[4]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 [17].

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

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

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

NoSASP GitHub Adapter: `https://github.com/dindonero/smartbugs-local-static-analysis` 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.

Ethereum Ethereum [5] may be perceived as a more robust version of the Bitcoin Blockchain which enables a broader range of applications. 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.

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"*[31].

Solidity [30] is the world's first Turing complete programming language [12] 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.

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.

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 DASP Top 10 taxonomy[23]. All of SmartBugs' tools reported vulnerabilities were mapped into one of these categories: Reentrancy, Access control, Arithmetic Issues, Unchecked Low Level Calls, Denial of Service (DoS), Bad Randomness, Front Running, Time manipulation, Short Addresses, Unknown Unknowns.

SmartBugs The academic community has been working on developing automated analysis tools to find and eliminate vulnerabilities in smart contracts [20, 34, 35, 11, 15, 27, 21]. 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 [21, 9].

Its code[8] is written in Python 3 and the tools are run using Docker images. These images can be found on the Docker Hub[9] 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.

SmartBugs' authors gathered 35 possible suitable tools from a survey[6] and through research. From those 35 state-of-the-art analysis tools, only the following 10 were picked to be integrated into SmartBugs' environment: HoneyBadger, Maian, Manticore, Mythril, Osiris, Oyente, Securify, SmartCheck, Solhint.

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 Solidity smart contracts with 193 manually labeled vulnerabilities, categorized into the DASP 10 taxonomy which can be used to test the precision of analysis tools. And the SB^{WILD} dataset which 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[10].

[8]SmartBugs (including $SB^{CURATED}$): https://github.com/smartbugs/smartbugs

[9]Docker Hub: https://hub.docker.com/

[10]`https://etherscan.io/`

It was performed an experimental analysis [8] using seven tools on the $SB^{CURATED}$ dataset with the purpose of calculating the tools' ability to identify vulnerabilities in 69 contracts. 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 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.

SARIF 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 stands for Static Analysis Results Interchange Format. It originated at Microsoft, and is now a standard being developed under OASIS[11] (Organization for the Advancement of Structured Information Standards). The technical committee has members from several static analysis tool vendors and large-scale users [1].

SARIF is 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[12], Clang Static Analyzer[13] and Pylint[14]) [2, 19].

The SARIF's standard format [26] is composed of three main root keys: version, $schema and runs. The runs' key is the array that contains all the information related to the analysis where each object corresponds to a different static analysis tool and its fields are divided into categories: The analysis results, composed of the keys *artifact*, *invocations*, *results*. And the analysis metadata, composed of the *tool* key [18].

The analysis results details the information about the analysis itself. The *artifacts* is a list describing all files examined by the tool (even if results were not detected) which must not start with a backslash and must be relative to the repositories root. The *invocations* describes the invocation information of the analysis, mainly the when and the how it was analyzed. Finally, the *results* array contains the results yielded by the static analysis tool. Each object contains a defect reported by the analysis. It is composed by the *ruleId*, the *message*, the *level* (can be warning, error, note or none) and the *locations* array.

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.

SASP The Static Analysis Server Protocol (SASP) [1] 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.

GitHub GitHub is a web hosting applications for collaborative software development projects. 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.

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 CI. GitHub Actions are event driven, meaning a developer can specify a series of commands to be run after a specific event has occurred. It 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.

One study [36] has shown that teams working on

[11] `https://www.oasis-open.org/`
[12] CogniCrypt: `https://www.eclipse.org/cognicrypt/`
[13] CSA: `https://clang-analyzer.llvm.org/`
[14] Pylint: `https://www.pylint.org/`

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.

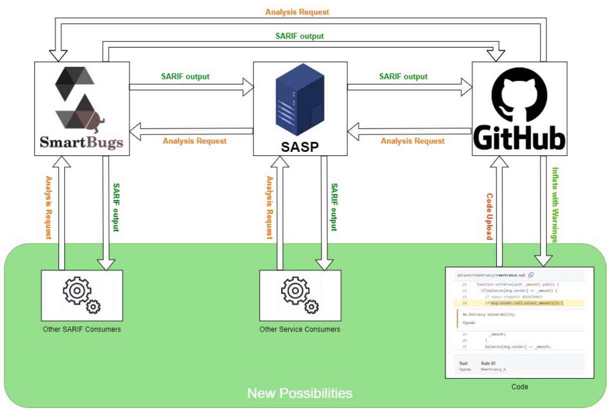
3. Implementation

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 lacks 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 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 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 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.

SARIF Converters The SARIF converters for each tool were written in Python based on the *sarif_om*[15] module developed by Microsoft on GitHub.

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. Furthermore we developed a sarif converter from sarif_om to SARIF JSON format which had already been requested by multiple users on GitHub.

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 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.

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

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. This table can be found in the `smartBugs/src/output_parse/sarif_vulnerability_mapping.csv` file.

SASP We build SASP 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 2.

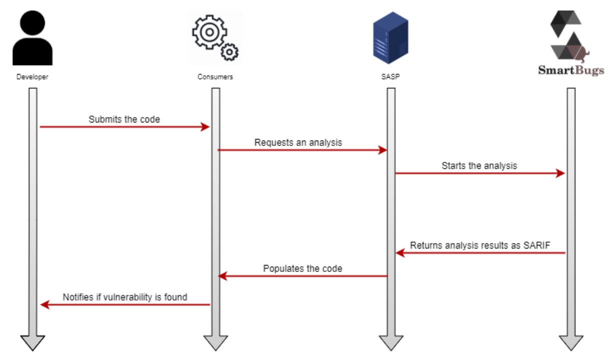


Figure 2: Example of a end consumer working with SASP to populate comments for an arbitrary code submitting

SASP was developed using Python 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.

GitHub Review Bot In contemplation of providing a mechanism for a straightforward interpretation of the results produced by these tools, we created an automated GitHub review bot 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. 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.

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. An example for this config file can be seen in Listing 1.

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.

Listing 1: Example of the file *.github/workflows/smartbugs.yml* that triggers SmartBugs analysis automatically at every pull request

```
name: "Run SmartBugs analysis"

# When the workflow is run
on:
 pull_request:

jobs:
 deployment:
  runs-on: ubuntu-latest
  steps:
   - name: Run SmartBugs Analysis
   uses: smartbugs/
     smartbugs-static-analysis@v2
   with:
    # Specify the tools
          tool: 'conkas oyente mythril'
```

NoSASP approach Alternatively to the SASP approach 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. Furthermore it is bounded by 14 GB of SSD storage which can constraint some projects on using this service.

False positives 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. We hope to provide an more embraceable methodology to ignore unwanted results so that a developer can focus on its code real security problems.

Conkas Conkas[16] is a symbolic execution-based static analysis tool for the Ethereum Virtual Machine (EVM). 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. This tool shows great promise as it is one of the less CPU consuming tools, with great accuracy metrics.

SmartBugs' 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. 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.

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. 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.

4. Results

Our work validation consisted 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

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.

Validation In this step, we used the public free SARIF validation tool provided by Microsoft[17] to ensure our standard correctness. This tool comes with a feature that checks as well for the GitHub ingestion rules so it can validate that the entire dataset's results are correctly formatted for display on the SARIF processing mechanism from GitHub.

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 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.

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

[17]SARIF Validation Tool

For this step we used and recommend the The SARIF Web Viewer[18], which is a tool that allows any developer to see a robust interface detailing the SARIF file uploaded by the user.

4.2. SASP Service

Functionality Firstly we tested if server's was 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.

Secondly, we sent multiple badly formed requests. The SASP service behaved exactly as expected, ignoring the requests when it was required. We also tested it against the directory transversal attack which was correctly mitigated by the algorithm provided by [10].

Finally, we decided to conduct a stress test to see how our server would behave when queried by multiple requests from different users. With this test we hope to deduce the approximate scalability of our developed solution with the purpose of preventing clogged or crashed services.

For a CPU intense analysis we combined 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 graph can be seen in Figure 3.

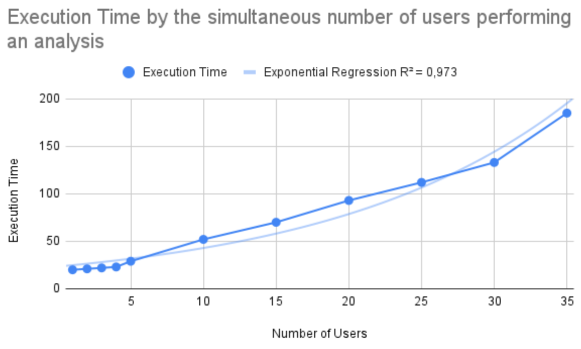


Figure 3: SASP stress testing with Mythril and Slither analysing SimpleDAO and Reentrancy

For our surprise, the SASP service is more scalable than we had anticipated. Up to 35 clients we get a linear regression as the time increases constantly as per number of clients requesting our service. However when we get to 40 simultaneous analyses, Docker service starts to crash and consequently SASP stops all current analysis so that it may recover from the crash. There is an existing issue on GitHub that is being assessed, contemplating this Docker bug [29] and it keeps us from reaching further conclusions.

4.3. GitHub Adapter

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, 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.

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 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.

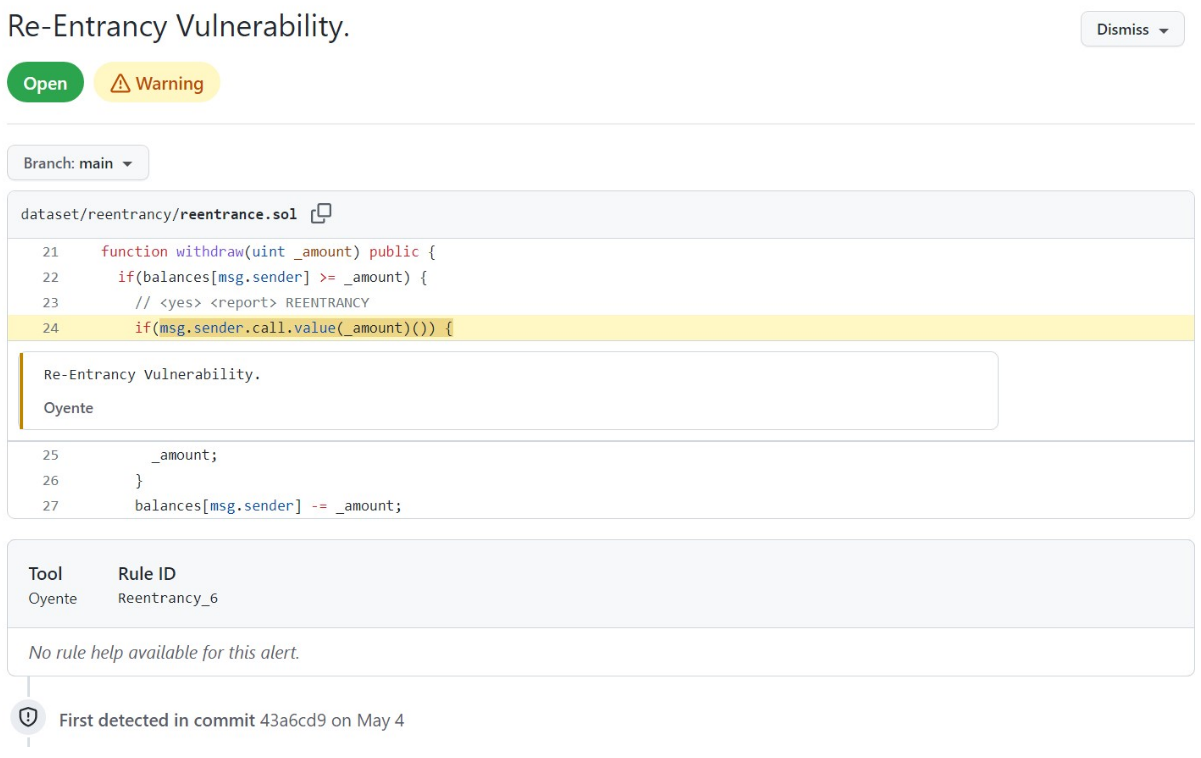
Intuitiveness At last, we will discuss if the comments and the system as a whole are intuitive enough for any programmer to be able to abstract from the analysis and only focus on coding the smart contract itself.

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. With SmartBugs the number of outputs was significantly reduced from (the number of tools) x (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

[18]SARIF Web Viewer


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

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. Our final result can be seen in Figure 4

5. Conclusions

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 paper 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.

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. For the future we would like to see other static analysis tools playing SmartBugs' role in the diagram in Figure 1.

References

[1] P. Anderson. Static analysis results: A format and a protocol: Sarif & sasp, 2018.

[2] P. Anderson, L. 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)*, pages 71–72. IEEE, 2019.

[3] A. M. Antonopoulos. *Mastering Bitcoin: unlocking digital cryptocurrencies.* " O'Reilly Media, Inc.", 2014.

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

[5] V. Buterin. Ethereum whitepaper. 2013.

[6] 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)*, pages 69–78. IEEE, 2019.

[7] 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.

[8] 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*, pages 530–541, 2020.

[9] 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*, pages 1349–1352, 2020.

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

[11] 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*, pages 243–269. Springer, 2018.

[12] N. Gruhn. What makes a programming language turing complete?, 2019.

[13] 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*, pages 188–207. Springer, 2020.

[14] L. Hollander. The ethereum virtual machine — how does it work?, 2019.

[15] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In *Ndss*, pages 1–12, 2018.

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

[17] K. J. Kistner. itoken duplication incident report, 2020.

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

[19] 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.

[20] 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*, pages 254–269, 2016.

[21] A. P. C. Monteiro. A study of static analysis tools for ethereum smart contracts. 2019.

[22] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008.

[23] NCCGroup. Decentralized application security project (or dasp) top 10, 2018.

[24] 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*, pages 653–663, 2018.

[25] M. Nofer, P. Gomber, O. Hinz, and D. Schiereck. Blockchain. *Business & Information Systems Engineering*, 59(3):183–187, 2017.

[26] Oasis. Static analysis results interchange format (sarif) version 2.1.0.

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

[28] A. Rosic. What is blockchain technology? a step-by-step guide for beginners.

[29] sheridp. Container.wait with timeout raises connection error · issue 1966 · docker/docker-py, 2018.

[30] Solidity. Solidity documentation.

[31] N. Szabo. Smart contracts: building blocks for digital markets. *EXTROPY: The Journal of Transhumanist Thought,(16)*, 18(2), 1996.

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

[33] R. O. Team. Cover infinite mint exploit, 2020.

[34] 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*, pages 9–16, 2018.

[35] 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*, pages 67–82, 2018.

[36] 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*, pages 401–405. IEEE, 2014.