BENCHMARKING ETHEREUM SMART CONTRACT STATIC ANALYSIS TOOLS

An Undergraduate Research Scholars Thesis

by

TERRELL CADE FORD

Submitted to the LAUNCH: Undergraduate Research office at
Texas A&M University
in partial fulfillment of requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Faculty Research Advisor: Dr. Jeff Huang

May 2022

Major: Computer Engineering (Computer Science Track)

Copyright © 2022. Terrell Cade Ford.

RESEARCH COMPLIANCE CERTIFICATION

Research activities involving the use of human subjects, vertebrate animals, and/or biohazards must be reviewed and approved by the appropriate Texas A&M University regulatory research committee (i.e., IRB, IACUC, IBC) before the activity can commence. This requirement applies to activities conducted at Texas A&M and to activities conducted at non-Texas A&M facilities or institutions. In both cases, students are responsible for working with the relevant Texas A&M research compliance program to ensure and document that all Texas A&M compliance obligations are met before the study begins.

I, Terrell Cade Ford, certify that all research compliance requirements related to this Undergraduate Research Scholars thesis have been addressed with my Research Faculty Advisor prior to the collection of any data used in this final thesis submission.

This project did not require approval from the Texas A&M University Research Compliance & Biosafety Office.

TABLE OF CONTENTS

Page

ABSTRACT 1

DEDICATION 3

ACKNOWLEDGEMENTS 4

NOMENCLATURE 5

CHAPTERS

1. INTRODUCTION 6
   - 1.1 Ethereum 7
   - 1.2 Decentralized Finance 8
   - 1.3 Smart Contract Static Analysis 8
   - 1.4 Related Work 9

2. METHODS 10
   - 2.1 SmartBugs 10
   - 2.2 Runtime Analysis 11
   - 2.3 Detection Rate Analysis 11
   - 2.4 Tool Ranking 12
   - 2.5 Usability Score 12

3. RESULTS 14
   - 3.1 Runtime Analysis 14
   - 3.2 Detection Rate Analysis 19
   - 3.3 Tool Ranking 24
   - 3.4 Usability Score 27

4. CONCLUSION 29
   - 4.1 Singular Tool Use 29
   - 4.2 Multiple Tool Use 30

REFERENCES 31

APPENDIX A: DURATION PLOTS OF ANALYSIS TOOLS 32

APPENDIX B: DURATION PLOTS OF ANALYSIS TOOLS WITH CERTAIN METRICS .. 35

ABSTRACT

Benchmarking Ethereum Smart Contract Static Analysis Tools

Terrell Cade Ford
Department of Computer Science & Engineering
Texas A&M University

Research Faculty Advisor: Dr. Jeff Huang
Department of Computer Science & Engineering
Texas A&M University

This project benchmarks the operation of existing Ethereum smart contract static analysis tools. This is to support the proliferation of tools which allow developers to screen their Ethereum smart contracts for security vulnerabilities and determine what tool or tool suite would be most appropriate for bulk scanning of the entire Ethereum decentralized finance (DeFi) space. This is achieved by comparing the relative performance of several separate static analysis tools on various curated smart contracts. Each tool is made to analyze a list of smart contracts which have known vulnerabilities of various categories dispersed throughout. The resulting output of each static analysis tool is analyzed in several key ways. First, the general runtime of the tool is measured for each input smart contract. This is broken down into metrics such as time taken per line of code, time per kilobyte of file size, and time vs code complexity. Second, the number of vulnerabilities detected by each tool is taken into account. Each tool is capable of detecting different types of vulnerabilities with substantial overlap between tools. The capabilities of the tools are evaluated and scored based on the number of total vulnerabilities found, as well as how many different types of vulnerabilities are capable of being found. Finally, the general accuracy

of each tool is compared. The number of false positives and false negatives for each vulnerability category and tool are displayed and compared. Added together, these benchmarking categories are combined into an overall usability score for each tool. This usability score is employed to determine what tool or set of tools could be used to screen individual smart contracts, as well as bulk scan the entire DeFi space.

DEDICATION

I dedicate this work to my parents, friends, and mentors who have shaped me into the man I am today. I wouldn't be here without your influence on me. Specifically, to my dad, you are an inspiration and a role model I strive to emulate every day.

ACKNOWLEDGEMENTS

Contributors

Thank you, Dr. Jeff Huang, for your experience, patience, and assistance of me during this venture. You have proved to be invaluable several times over through your prompt email responses alone. Additionally, you have been an excellent motivator, inspiring our O2 lab group to continue learning about and working on new things in this emerging smart contract space. To the other members of our lab, Shivam, Bozhen, Austin, and Tien, keep up the great work!

The analysis platform *SmartBugs*, used for this manuscript, was developed in part by Drs. Joao F. Ferreira, Thomas Durieux, Pedro Cruz, and Rui Abreau, among others. Their tool and corresponding paper, *SmartBugs: A Framework to Analyze Solidity Smart Contracts*, made this solo undergraduate research project possible.

Funding Sources

Undergraduate research was supported by Dr. Jeff Huang at Texas A&M University.

NOMENCLATURE

BTC	Bitcoin
DAO	Decentralized Autonomous Organization
DeFi	Decentralized Finance
EVM	Ethereum Virtual Machine
ETH	Ethereum
NFT	Non-Fungible Token

1. INTRODUCTION

With the constant flood of easy money from the Federal Reserve, the purchasing power of the US dollar, the global reserve currency, is dwindling with no end in sight. Some market observers suggest the stock market appears manipulated by the wealthy to continually reach all-time highs to make themselves wealthier. Global tensions are also rising between countries large and small, seen plainly by the invasion of Ukraine by Russia. This is causing wild swings in currency and foreign exchange markets across the globe. It's in this shadow of uncertainty about the future of finance and the economy that many have turned to a new, digital sector of finance—cryptocurrency. Every day it seems like another company is interested in offering digital currency to their customers or interacting with the space in some fashion. All the time there are headlines about new trading platforms being erected or more companies accepting Bitcoin (BTC), Ethereum (ETH), or some other cryptocurrency as payment. Innovation in the crypto sector is constantly producing new ways to interact with various coins and projects, and many institutions want into the sector for a variety of reasons. Whatever the motivation, people are flooding into digital currencies at a breakneck speed, possibly faster than the adoption of the internet.

With the constant influx of people comes a steady source of people who are unaware about how cryptocurrencies work and inevitably get tripped up in an undeniably new and unique environment. Specifically, there are several risks to owning cryptocurrencies which can catch new and sometimes veteran users unawares. First, individuals can lose their crypto by losing the private keys which control their virtual currency wallet. Whether through misplacement or accidental deletion, many people have already lost life savings and fortunes in digital currency.

Second, someone can fall victim to one of numerous scams online. A popular one is in the form of a fake giveaway, where a user is enticed to send currency to an address with the promise of receiving more in return. Obviously, none is returned. Finally, there are hacks and vulnerabilities in the programs which run the network itself. Through some fault of the coder or custodian, funds can be stolen by hackers on various platforms and different networks.

Bleeding edge innovation is so named because of the metaphorical blood it produces. Moreover, numerous projects in the decentralized finance space have been outright hacked or had legitimate features taken advantage of to the detriment of the users who participate in them. Therein lies a problem and a hypothetical question. Are there ways to reduce the number of dangers in the cryptocurrency space so that it’s less hazardous for participants and more forgiving to newcomers? The last problem, vulnerabilities in the programs and networks which underlie the digital currency space, is quite complex and is the focus of this thesis. There already exist several tools to analyze code from these projects, but many of the tools are extraordinarily slow to run, generate many false positives and false negatives, and lack the capability to detect certain categories of vulnerabilities. This project first benchmarks the runtime of smart contract security analysis tools and afterward, categorizes how accurate they are in terms of the number of actual vulnerabilities detected. Finally, the project aims to assess what tool or suite of tools would be most appropriate for bulk scanning individual smart contracts and the entire Ethereum DeFi space.

1.1 Ethereum

Ethereum is a community-run technology powering the cryptocurrency and thousands of decentralized applications [7]. It is a borderless, peer-to-peer network that allows a user to move money and make transactions directly with another user—no need for a third-party intermediary

to handle the exchange. ETH requires no personal information to hold and maintain a wallet, and users can acquire Ethereum without owning a bank account. Ethereum is not a company. It's a community native to the Internet and is available to anyone with a network connection. Its disassociation with governments and countries means that any person or group can use Ethereum regardless of their nationality. Ethereum is home to innumerable projects and startup enterprises with various goals and use cases. These use cases are generally organized into three categories: Decentralized Finance (DeFi), Non-fungible tokens (NFTs), and decentralized autonomous organizations (DAOs).

1.2 Decentralized Finance

Decentralized finance is a subfield of cryptocurrency which deals with financial transactions on a decentralized blockchain [6]. Smart contracts are the vehicle which drive DeFi by introducing a way to add custom functionality to the public blockchain. These contracts run on the Ethereum Virtual Machine (EVM) in the form of compiled bytecode. Applications have been developed to read either these bytecodes, or the Solidity language source code which most smart contracts derive from. They can analyze these contracts for issues which would make the contract behave in a way that was unintended to the programmer. Since contracts become immutable once they are submitted to the blockchain, great care must be taken to ensure that either the contract is bug-free before submission, or that the owner is notified afterward to remove funds from a vulnerable contract.

1.3 Smart Contract Static Analysis

This research project is focused on the field of decentralized finance. Specifically, Ethereum smart contract static analysis tools have been benchmarked in terms of their runtime efficiency and number of security vulnerabilities detected. The specific static analysis tools

include: Mythril, Slither, SmartCheck, Securify, HoneyBadger, Osiris, and Conkas, as well as others [4]. The types of vulnerabilities being logged include access control, arithmetic, bad randomness, and denial of service, as well as reentrancy, time manipulation, and unchecked calls, among others [8]. Each of these tools have been tested to determine which of them runs most efficiently in a variety of categories. These primarily include execution speed, number of vulnerabilities found for a given type, fewest false positives raised, or a combination thereof.

1.4 Related Work

Little has been mentioned in papers except general complaints regarding the relative slowness of some algorithms at detecting vulnerabilities. Therefore, it seems reasonable to direct a study toward benchmarking each major smart contract analysis tool in terms of runtime, number of bugs found, and the accuracy of those findings. Analyses like in Perez's "Smart Contract Vulnerabilities" do an excellent job of detailing what analysis protocols are capable of detecting which vulnerabilities, but few describe how feasible it would be to efficiently uncover all potential bugs in submitted contracts [1]. A useful tool for analyzing smart contracts automatically has been a library referred to as *SmartBugs*. This library executes smart contract analysis programs in self-contained Docker instances and times their execution speed as well. This library has been exceptionally helpful in determining the performance metrics of each analysis tool when analyzing vulnerable smart contracts.

2. METHODS

This research question has been tackled by downloading several major static analysis tools for Ethereum smart contracts. These include tools such as Osiris, Mythril, Securify, Slither, and several others [4]. The relative performance in terms of runtime and smart contract size has been compiled into spreadsheets to compare various tools. There are several smart contract analysis tools downloaded in Docker images to a Linux virtual machine which have been run on Solidity source code. A quantitative research methodology has been employed which recorded the total runtime distribution of each tool on a curated list of smart contracts. An additional analysis has also been done between smart contract platforms in terms of their relative ability to detect smart contract vulnerabilities and the types of vulnerabilities each tool is capable of detecting at all.

2.1 SmartBugs

SmartBugs has been an exceedingly useful tool for systematically benchmarking smart contract analysis tools. When running *SmartBugs* with a selected smart contract tool, several things happen. First, *SmartBugs* automatically downloads a Docker image of the smart contract analysis tool to run it in a contained environment. Then, *SmartBugs* executes the analysis tool on a specified dataset of annotated vulnerable smart contracts. The execution script is specified in a .yaml file in the *SmartBugs* repository, which makes the tool easily extensible to future smart contract analysis tools. After the tool has finished its analysis, which varies from seconds to minutes, several results files are generated. A .json file is generated which contains the parsed output of the analysis tool into a common format which can be read from and analyzed later. The tool has numerous smart contracts which have been analyzed and marked with the location and

type of each vulnerability contained. These smart contracts are referred to in this paper as curated contracts to differentiate them as having known vulnerabilities.

2.2 Runtime Analysis

There are two primary quantitative methodologies employed in benchmarking Ethereum smart contracts. First, the runtime of each analysis tool is recorded for each curated smart contract. Runtime analysis is broken down into multiple key metrics. One metric is the raw number of seconds taken by each tool on each smart contract. Three other metrics are related to the runtime of each tool divided by the file size of the contract in question, the number of lines of code in the contract, and the number of keywords in the selected contract. Selected runtime comparisons are collated into graphs directly comparing the time taken by verification tools on each curated contract. These runtimes versus file size, code lines, and keywords are referred to as "normalized" runtime because they divide the raw runtime by the metric measured against.

2.3 Detection Rate Analysis

The second quantitative analysis method compares the number of actual vulnerabilities detected by each static analysis tool. Vulnerabilities detected are put in terms of the accuracy rate of bug detection, the number of false positives, and the number of false negatives. The inclusion of false positives as well as false negatives is important to the assessment of these static analysis tools because of the purpose they serve. False positive detections erroneously alert the programmer or user that a vulnerability exists where there is none. This causes multiple issues. First, it will waste the time of the programmer in investigating a vulnerability which does not exist. Second, if too many false positives exist, the programmer will tend to ignore the warnings of the analysis tool and potentially miss a real detection at a later date. False negatives however, are worse. If a programmer is never alerted to the presence of a vulnerability, the capacity exists

for a programmer to miss the vulnerability and publish the contract with an existing bug. Later, a malicious user may take advantage of the contract and steal the cryptocurrency contained within. The detection accuracy is compared across each tool for multiple vulnerability categories. Some of these categories are: Reentrancy, Arithmetic, Unchecked low-level calls, Access control, Time manipulation, and Other. Each of these categories have wildly varying detection accuracies between the tools in question.

2.4 Tool Ranking

This wildly varying accuracy is the basis for a subjective ranking of the static analysis tools. Each tool is very different in its accuracy and ability to detect types of vulnerabilities. Some tools also generate many more false positives than others. Some tools are unfortunately entirely incapable of detecting certain types of vulnerabilities. In order to take this into account, the ability of each tool to detect a vulnerability category is compared against each other tool. The best tool to use for each vulnerability category is identified in a table. Tools incapable of detecting vulnerabilities are given no score, whereas tools able to detect them are ranked based on their relative performance against each other. This analysis relies on the vulnerability detection rates and runtime analysis of the previous tests, using both as a means to justify the ranking of each tool in each category.

2.5 Usability Score

The results of the previous three analyses are combined into a usability score. This score is generated through a decision matrix. The matrix is intended to do two things. It will help approach the selection of a single best analysis tool from a logical viewpoint rather than a subjective or intuitive one. It will also weigh a variety of important factors, some of which include raw runtime, normalized runtime of multiple types, detection accuracy for vulnerability

categories, and the ranking of each tool for multiple vulnerability types. The final usability score will be the primary factor used to answer the paper's research question: What tool or tool suite would be most appropriate for bulk scanning the entire Ethereum DeFi space?

3. RESULTS

3.1 Runtime Analysis

Excel was used once all duration and vulnerability analysis was completely finished computing with each smart contract tool. The primary means of analysis was runtime comparison between different tools. Graphs were generated with Excel to visualize the runtime differences between each smart contract on a raw and normalized basis. As seen in Figure 3.1, Conkas, HoneyBadger, and Mythril had some of the least predictable runtimes of the group, with the average runtime of Conkas being the slowest of the group.

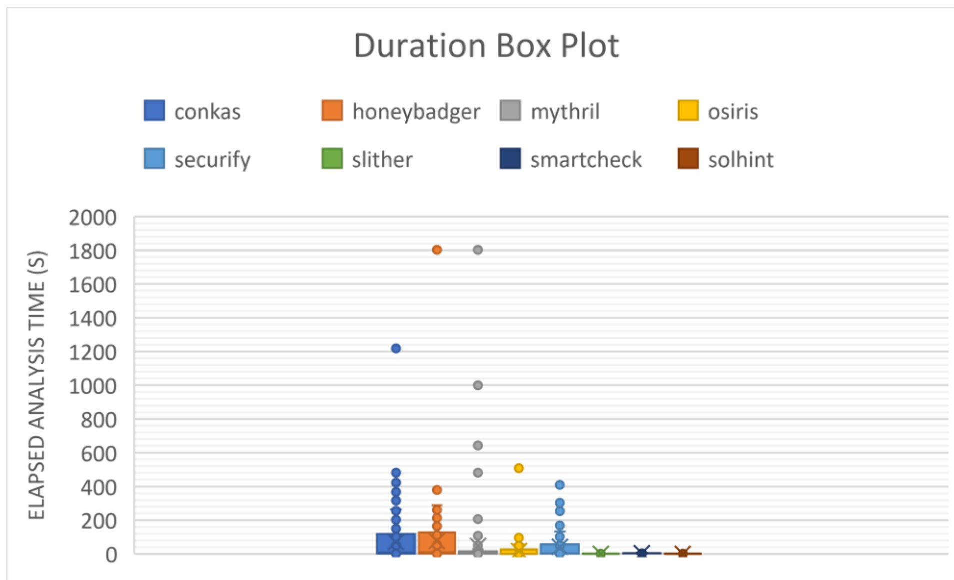

Figure 3.1: A plot of the duration (in seconds) of each tool on each contract.

Above is a box plot of the duration of each tool on each curated smart contract in raw seconds. This box and whisker plot identifies the mean elapsed time, upper and lower quartiles, and outlier runtimes. The average runtime is somewhat low, but certain runs take substantially longer than others. A ninth smart contract tool, Manticore, was too slow to be viably analyzed,

taking an average of over twenty minutes per analyzed smart contract. For normalization, each raw runtime on a particular smart contract was divided by the file size of the contract. Other methods were considered for normalization, but none produced any meaningful difference from the normalization method employed originally. Additionally, large normalized duration differences between certain smart contract analysis tools made it difficult to accurately graph each smart contract tool side-by-side. Therefore, a log scale chart was used in Figure 3.2 to compress widely varying data. Pictured below is the logarithmic normalized duration plot of each static analysis tool. This plot is normalized by dividing the runtime of each contract analysis by the number of bytes contained in the contract. A log scale is used to make visible the magnitudes-wide average runtime differences between various analysis tools.

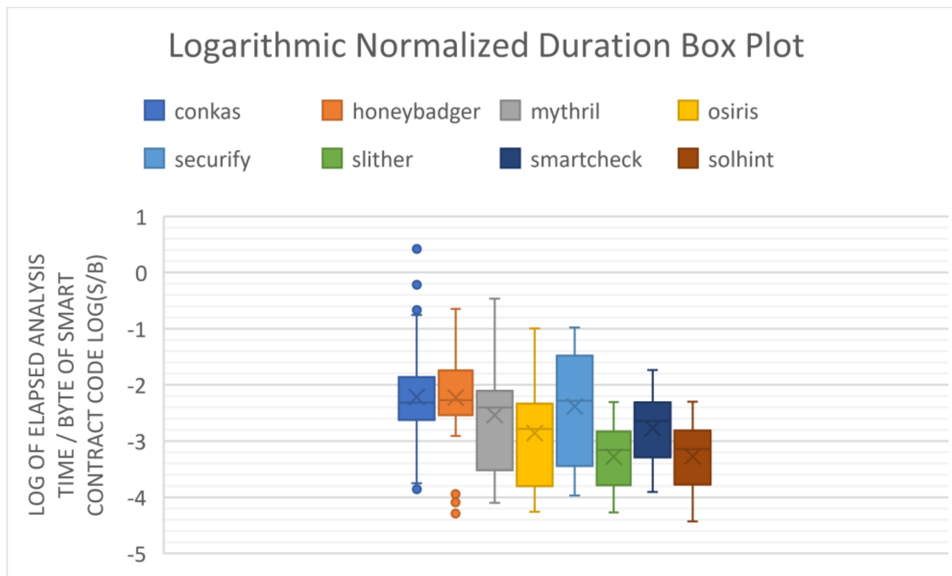

Figure 3.2: Relative runtime performance of eight smart contract analysis tools normalized by file size.

Excel was used to create scatter plots of various smart contract runtimes versus the number of lines of code, number of keywords in the contract, and file size. In particular, the

keyword analysis summed the total number of Solidity code keywords, such as “function,” “event,” “constructor,” and “modifier.” Several other variations of the chart are in Appendix A. Different methods of normalization for each contract runtime are in Appendix B.

The fastest smart contract tool was Slither, averaging below 1ms per byte of smart contract code. The slowest was Conkas, at 36ms per byte of code. SmartCheck was nearly as fast as Slither, but its functionality in terms of the number of vulnerabilities it could detect was somewhat less than Slither. Mithril’s performance was somewhat disappointing despite its great reputation. It ran more than 22 times slower than Slither, but was capable of detecting only slightly more varied vulnerabilities. If a recommendation was to be made solely in terms of the most time-efficient analysis tool, Slither and Solhint would be standout winners. The average runtimes of each smart contract analysis tool on each contract are reproduced below in Figure 3.3.

	Conkas	HoneyBadger	Mythril	Osiris	Securify	Slither	SmartCheck	Solhint
Average Runtime (Normalized)	35.87 ms/B	23.84 ms/B	19.81 ms/B	8.01 ms/B	19.21 ms/B	0.89 ms/B	3.08 ms/B	0.92 ms/B
Standard Dev. of Runtimes	199 ms/B	39.73 ms/B	60.48 ms/B	17.55 ms/B	27.00 ms/B	0.82 ms/B	3.02 ms/B	0.84 ms/B
Coefficient of Variation	555%	167%	305%	219%	141%	92%	98%	91%
Average Runtime vs. Slither	40.27	26.77	22.24	8.99	21.58	1.00	3.46	1.03

Figure 3.3: Table of normalized duration data for each smart contract tool.

As another interesting note, Slither, SmartCheck, and Solhint had the least variance of runtime based on a comparison of the coefficients of variation (standard deviation divided by the mean). Consequently, because they had the quickest overall runtime, the somewhat lower variance amounts to much less of an actual time difference between the slowest and fastest

contract analysis than most other tools. Additionally, Conkas had the largest ratio between the standard deviation of each runtime and the mean runtime. It also had the highest average runtime when excluding Manticore. So, those compound to make Conkas the least predictable with its runtime, making Conkas the least desirable tool to use in terms of runtime analysis.

Another product of runtime analysis comes in the form of plotting each analysis tool's performance for each smart contract. There are three primary ways to plot each smart contract analysis duration. These have already been specified as file size, lines of code, and number of keywords. One of each of these types has been displayed below for visual comparison. Slither has been chosen as an example tool because it has one of the lowest coefficients of variation. Thus, a clearer trend can be depicted than with other tools. Figure 3.4 graphs the duration of Slither analysis versus file size. Figures 3.5 and 3.6 respectively graph versus lines of code and number of keywords.

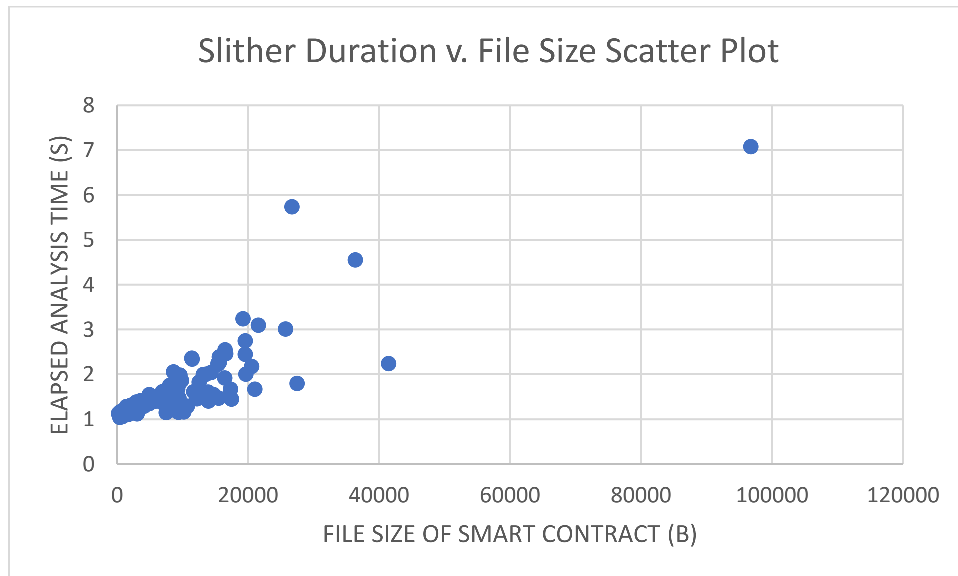

Figure 3.4: Slither's runtime relative to the size of the input code file in bytes.

Figure 3.5 is a plot of Slither's runtime on each contract per line of code in the given contract. Both graphs are visibly similar.

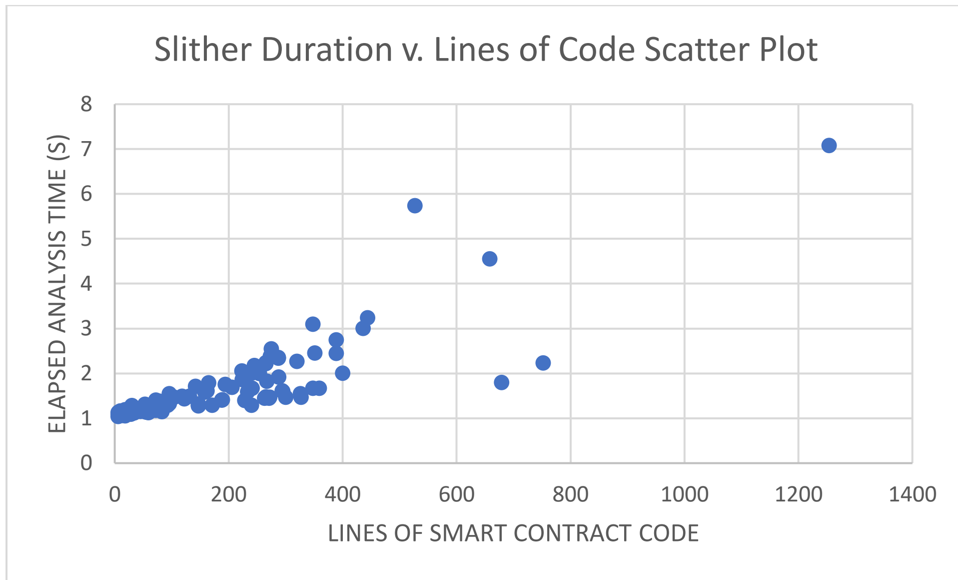

Figure 3.5: Slither's runtime relative to the number of lines of contract code.

Finally, the plot of Slither duration versus keyword count is given in Figure 3.6.

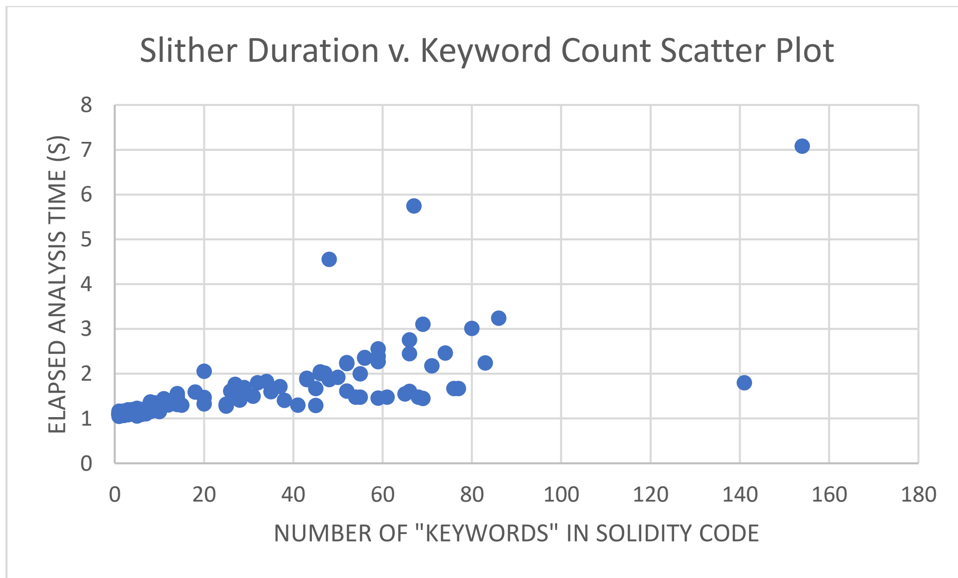

Figure 3.6: Slither's runtime compared to the number of keywords in each contract.

Again, Figures 3.4-3.6 have a clear similarity. Although, there are still some distinguishable differences. The line of best fit is clearly different between each graph. Several other analysis tools have been plotted for a variety of normalization methods in Appendix B. An R squared analysis of each graph could be conducted on each to determine how well each normalization metric performs for Slither contracts. In that way, the best normalization method could be deduced. However, that has not been deemed necessary or useful enough in answering the project's overall research question: What tool or tool suite would be most appropriate for bulk scanning the entire Ethereum DeFi space? An R squared analysis of the results has therefore not been done.

3.2 Detection Rate Analysis

The second overarching type of quantitative analysis comes in the form of the relative detection rates of each static verification tool. This analysis was split into two primary parts. The first detection rate test is on the general accuracy of each tool for each vulnerability type. This test also includes the raw numbers and types of vulnerabilities detected compared to the total numbers of vulnerabilities existent in all the smart contracts. The second detection rate test is of the types and variety of false positives and false negatives identified by the analysis tools. For all detection rate tests, each vulnerability identified by a tool or curated smart contract is split into one of four classifications. If the vulnerability type and location identified by an analysis tool are the same as the type and location tagged in the curated smart contract, it is counted as a perfect match. If only one of the two criteria line up, it is deemed a partial match. If neither match, it's a false positive. Finally, if a tagged vulnerability has no associated perfect or partial match, it's counted as a false negative.

For the first detection rate test, each .json file generated by *SmartBugs* for analysis was scanned for the type and location of detected vulnerabilities. Each vulnerability identified by a tool was checked against all existing tagged vulnerabilities in a specific smart contract. This was done for the numerous curated smart contracts to obtain a total number of vulnerabilities. The results of this analysis are displayed below in Figure 3.7. It is important to note that perfect matches and partial matches were not weighted equally. For the purposes of displaying the data, partial matches were weighted as half of a match because only one of the type or location criteria were identified correctly.

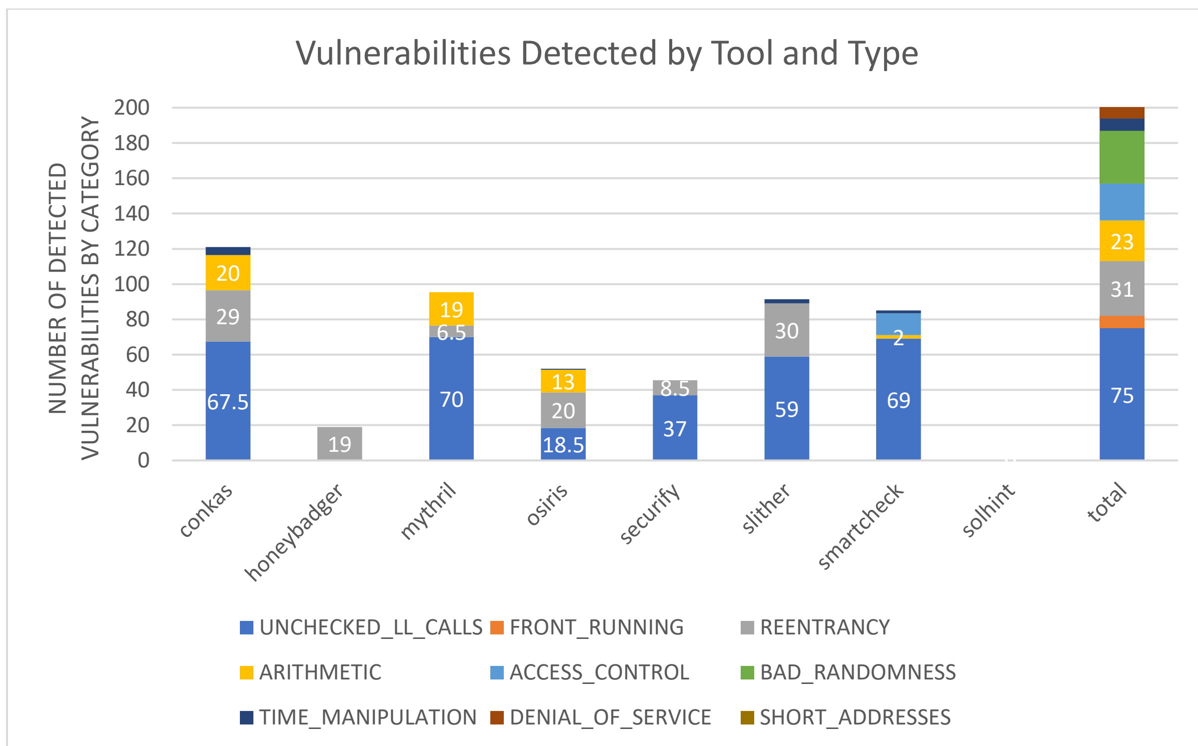

Figure 3.7: Vulnerabilities types and numbers detected by each tool.

In terms of the number of successfully identified vulnerabilities, Conkas found the most. Conkas, Mythril, Slither, and SmartCheck all were superior performers, doing much better than the remaining four analysis tools. Solhint was wholly unable to identify specific vulnerabilities or even line numbers on which they occurred. Instead, Solhint gave primarily syntactic and

indentation suggestions to the user. Overall, the most effective tool, Conkas, still only identified 111 vulnerabilities perfectly and 20 partially, out of the 202 opportunities. That's an overall effectiveness of about 60%, depending on how the partial identifies are interpreted. Unfortunately, no tool was capable of perfectly or partially identifying vulnerabilities arising from bad randomness, front running, denial of service, or short addresses. In particular, 30 of the vulnerabilities in the curated set were for bad randomness. The fact that none of the tools were able to identify the error at all is concerning.

The raw numbers of vulnerabilities detected in each category are summed up as an accuracy rate in Figure 3.8. The number of perfect matches plus half the number of partial matches is divided by the total number of tagged vulnerabilities across all contracts.

	Conkas	HoneyBadger	Mythril	Osiris	Securify	Slither	SmartCheck	Solhint
UNCHECKED_LL_CALLS	90%	0%	93%	25%	49%	79%	92%	0%
FRONT_RUNNING	0%	0%	0%	0%	0%	0%	0%	0%
REENTRANCY	94%	61%	21%	65%	27%	97%	0%	0%
ARITHMETIC	87%	0%	83%	57%	0%	0%	9%	0%
ACCESS_CONTROL	0%	0%	0%	0%	0%	0%	60%	0%
BAD_RANDOMNESS	0%	0%	0%	0%	0%	0%	0%	0%
TIME_MANIPULATION	64%	0%	0%	7%	0%	36%	21%	0%
DENIAL_OF_SERVICE	0%	0%	0%	0%	0%	0%	0%	0%
SHORT_ADDRESSES	0%	0%	0%	0%	0%	0%	0%	0%

Figure 3.8: Accuracy rate of each tool by vulnerability type.

As can be inferred from Figure 3.7, the unchecked low-level call and reentrancy vulnerabilities generally have good coverage across the gamut of static analysis tools. But, certain tools like SmartCheck stand out for their ability to detect vulnerabilities that no other platform can. Certain other tools should be praised for their high overall accuracy in various categories. Mythril had the highest accuracy in low-level calls. Slither was astonishingly accurate when screening for reentrancy vulnerabilities. These tests highlight the widely varying capabilities and focuses between tools focused on finding vulnerabilities in smart contracts.

The second detection rate test highlights the number of false positives and false negatives generated by each static analysis tool. The assessment of this is important to inform a user or smart contract programmer of what vulnerabilities may be overrepresented or missed when employing each analysis tool. The number of false negatives can be extrapolated from the data of Figures 3.7 and 3.8. However, the number of false positives by vulnerability type has not been directly displayed yet. In general, it is better for an analysis program to allow more false positives to appear than false negatives. This is because the potential for harm from missing a vulnerability is increased when a program fails to notify the user of a bug. However, sometimes the number of false positives can be excessive to the point of discouraging a user or programmer from investigating potential flagged vulnerabilities. With so many false notifications, people naturally tend to ignore them despite the possibility of one or multiple being valid. A table of the false positive and false negative vulnerability detections has been gathered from the output of each analysis tool on every curated smart contract. They are provided below as Figures 3.9 and 3.10.

	Conkas	HoneyBadger	Mythril	Osiris	Securify	Slither	SmartCheck	Solhint
UNCHECKED_LL_CALLS	2	75	0	55	3	14	0	75
FRONT_RUNNING	7	7	7	7	7	7	7	7
REENTRANCY	0	12	22	11	14	1	31	31
ARITHMETIC	1	23	1	10	23	23	20	23
ACCESS_CONTROL	21	21	21	21	21	21	0	21
BAD_RANDOMNESS	30	30	30	30	30	30	30	30
TIME_MANIPULATION	2	7	7	6	7	4	5	7
DENIAL_OF_SERVICE	7	7	7	7	7	7	7	7
SHORT_ADDRESSES	1	1	1	1	1	1	1	1
total	71	183	96	148	113	108	101	202

Figure 3.9: Raw number of false negative detections by tool and vulnerability type.

The number of false negatives, in light of the fact that there were 202 total tagged vulnerabilities, seems concerningly high. The fewest number of false negatives were logged by

Conkas. The situation is not quite as bad as it seems though, because the vast majority of detection misses come from the bad randomness and access control vulnerabilities. Nearly every tool had issues identifying those two categories. SmartCheck though, was fortunately able to catch all access control violations. If the access control and bad randomness categories are ignored, the total false negatives from Conkas come to a more reasonable 20 misses. Regrettably, when seen in the light of the number of false positives generated across tools, SmartCheck reveals itself to be far less appealing than initially thought. Figure 3.10 details the specific number of false positives across tools and vulnerability categories.

	Conkas	HoneyBadger	Mythril	Osiris	Securify	Slither	SmartCheck	Solhint
UNCHECKED_LL_CALLS	36	0	126	30	623	51	46	0
FRONT_RUNNING	0	0	0	0	0	0	0	0
REENTRANCY	340	0	80	20	403	115	4	0
ARITHMETIC	695	0	305	465	0	0	107	0
ACCESS_CONTROL	0	0	0	0	0	0	816	0
BAD_RANDOMNESS	0	0	0	0	0	0	0	0
TIME_MANIPULATION	108	0	0	17	0	6	0	0
DENIAL_OF_SERVICE	0	0	0	0	0	0	0	0
SHORT_ADDRESSES	0	0	0	0	0	0	0	0
OTHER	49	17	223	115	1468	531	1915	6577
total	1228	17	734	647	2494	703	2888	6577

Figure 3.10: Raw number of false positive detections by tool and vulnerability type.

In this assessment, SmartCheck, Securify, and Solhint displayed vastly more false positive vulnerability detections than the other five static analysis tools. Put in context, that is several times more clutter and false signaling than the remaining tools. Solhint is understandably an outlier however. It deals primarily with syntax and formatting issues in Solidity smart contracts, rather than stereotypical vulnerabilities. What was very interesting to realize in Figure 3.10 was that HoneyBadger logged virtually zero false positive detections. Every time a vulnerability was logged, it was either a perfect match or a partial match to a tagged vulnerability

somewhere in the smart contract. But with a tool only capable of detecting reentrancy and "other" vulnerabilities, such a low false positive rate seems to make sense. From above, the "other" category should be taken note of. That category deals primarily with smart contract issues which were not in any of the tagged categories. Some could have been legitimate detections of issues, but given that there is no way of verifying their applicability, they are counted instead as potential false positives.

3.3 Tool Ranking

This vulnerability analysis highlights the widely varying capabilities of each static analysis tool. However, beyond the question of accurately placing and detecting a vulnerability, some tools are simply incapable of detecting vulnerabilities. After reviewing the data output from each analysis tool on the smart contracts, the capabilities of each tool have been identified. In Figure 3.11, a matrix detailing which tool is able to catch which vulnerability has been devised. For each category mentioned in this matrix, those tools which are able to detect a vulnerability are assessed against each other for the relative accuracy and helpfulness of the tools in question. The relative positionings for each tool on each vulnerability category are recorded and factored in the final usability score at the end of this chapter.

Analysis Tool Capabilities Matrix

	conkas	honeybadger	mythril	osiris	securify	slither	smartcheck	solhint
Re-entrancy	●	●	●	●	●	●	●	
Arithmetic	●		●	●			●	
Unchecked Low Level Call	●		●	●	●	●	●	
Access Control							●	
Time Manipulation	●			●		●	●	
Other	●	●	●	●	●	●	●	●

Figure 3.11: Capability matrix of the potential for each tool to detect vulnerabilities.

This capabilities matrix details the respectively wide or narrow range of vulnerabilities each tool can scan for. Some programs are technically capable of uncovering each vulnerability type, as in the case of SmartCheck, but are not necessarily competent at detecting everything. Some tools, like Solhint, are more of a syntax and formatting focused tool with only a minor ability to point out errors in code. In the ranking of each tool, there were a few standouts. Conkas, SmartCheck, and Slither ranked consistently well in detecting smart contract issues. Even though their accuracy rate is lackluster in certain categories, other tools often manage far less or far worse in terms of execution speed and number of false positives and negatives. It should be noted that these rankings are somewhat subjective and are used as only one of multiple factors in the final usability score at this chapter's end.

Starting with Reentrancy, the tool deemed most useful for detecting that vulnerability was Slither. This came down primarily to the accuracy of its algorithm in finding reentrant bugs. Slither had a 97% accuracy rate, with 115 false detections. It only missed a single reentrancy bug among the 31 examples. It was also the fastest tool on the metaphorical static analysis block. Osiris was a close second, with 11 false negatives and 20 false positives. Conkas, though it had no false negatives, logged over 300 false positive reentrancy detections. The worst performer was definitely SmartCheck. There were no accurate detections of reentrancy from that tool. The few detections were all false positives. Making the tool capable of declaring a reentrant bug did not improve its functionality at all. The remaining ranks can be gathered from Figure 3.12.

The most common vulnerability across all smart contracts was unchecked low-level calls. Here SmartCheck redeems itself by identifying all possible low-level call bugs with a 92% identification accuracy and only 46 false positives. It also stood out as having a very fast overall runtime. Mythril also found all low-level call vulnerabilities, but it logged over 120 false positives. The third-best tool was Conkas. It missed two bugs and had a 90% overall accuracy. The worst tool for unchecked calls was Osiris.

The remaining tools have been ranked across the following vulnerability types: arithmetic, access control, and time manipulation. None of the other categories were used because all tools were equally ineffective at identifying errors or bugs of those types. The ranking of each tool in this section has been reproduced in Figure 3.12. Each first-place rating has been awarded seven points, with second placing getting six, and so on. Eighth place or a tool which was incapable of detecting a vulnerability type was given a score of zero for that category. The final score of each program has been calculated and displayed at the bottom of Figure 3.12.

	Conkas	HoneyBadger	Mythril	Osiris	Securify	Slither	SmartCheck	Solhint
REENTRANCY	3rd	4th	5th	2nd	6th	1st	7th	
ARITHMETIC	2nd		1st	3rd			4th	
UNCHECKED_LL_CALLS	3rd		2nd	6th	5th	4th	1st	
ACCESS_CONTROL							1st	
TIME_MANIPULATION	1st			4th		2nd	3rd	
total	23	4	16	17	5	17	24	0

Figure 3.12: Ranking of various analysis tools on several vulnerability types.

SmartCheck and Conkas revealed themselves to be excellent tools when ranked against each other analysis platform. Slither, Osiris, and Mythril were also good performers, with HoneyBadger, Securify, and Solhint falling far behind. SmartCheck in particular certainly benefited from being the only tool capable of detecting access control bugs in Solidity code. In light of the numerous false positives in the access control category, its bug detection was only slightly better than guessing. If just the two most common vulnerability categories of reentrancy and low-level calls were isolated, the best tools would have been Slither followed by Conkas and Mythril. The various ways of interpreting the program rankings highlights a need for a final overall usability score which takes into account all the analyses performed thus far.

3.4 Usability Score

Sections 3.1 through 3.3 are combined into an overall usability score to determine the most capable analysis tool. This usability score will be determined using a decision matrix to consider all relevant factors which are important in an Ethereum smart contract static analysis tool. These factors in order are: average runtime, vulnerability detection accuracy, number of false positives, number of false negatives, and finally their subjective ranking against each other. The ranking of tools against each other has been included to reward tools which are capable of detecting certain vulnerability types that others are not able to identify. The rows of the decision matrix are each of the eight potential options for selection of a best analysis tool. The five

columns will be each of the five factors, weighted according to how important they are to their goal of efficiently analyzing Ethereum smart contracts.

Each column has been given a weight between one and three. A one is a factor of low priority, something that is helpful but not strictly necessary. A weight of three is something that is crucial to the subject of effectively analyzing smart contracts. A program with a short runtime is helpful in screening for bugs because it causes fewer delays, but it's not specifically required. That category earns a one. The number of false negatives produced is the most important metric influencing analysis tools' effectiveness. A missed vulnerability could be exploited later after smart contract deployment and thus earns a three. The capacity to which each tool performs for each criterion is weighted between one and five. Ones are not performant or have incredible deficiencies, whereas fives are maximally efficient at the given category. The overall capability matrix is below in Figure 3.13.

	1: Runtime	2: Overall Accuracy	2: False Positives	3: False Negatives	2: Ranking	total
Conkas	1	5	3	4	5	39
HoneyBadger	2	1	5	1	2	21
Mythril	2	4	4	4	4	38
Osiris	3	2	4	2	4	29
Securify	2	2	2	3	2	23
Slither	5	4	4	3	4	38
SmartCheck	4	4	2	3	5	35
Solhint	5	1	1	1	1	14

Figure 3.13: Decision matrix for each analysis program over five categories.

Looking at the results, the miniscule margin of victory between Conkas, Slither, and Mythril is quite incredible. The weights and values for each category were chosen before calculating the total score to prevent bias. So, according to an analysis of runtime, accuracy, false positive and negative detections, and comparative ranking, Conkas is the most capable static analysis tool.

4. CONCLUSION

This research has been conducted to benchmark the operation of existing Ethereum smart contract static analysis tools. This is to support the proliferation of tools which allow developers to screen their Ethereum smart contracts for security vulnerabilities and determine what tool or tool suite would be most appropriate for bulk scanning of the entire Ethereum decentralized finance (DeFi) space. If smart contract owners are to be notified in the future of vulnerable contracts, a robust system must be designed to efficiently analyze smart contract source code or bytecode either before or after upload to the EVM. In particular, Conkas, Slither, and Mythril have been identified as three solid choices of static analysis tools which can be used to detect a plethora of vulnerability types. While many tools leave a lot to be desired in certain aspects, there is nothing stopping an Ethereum smart contract developer from using only one static analysis tool to screen for bugs. Ultimately, these tools will be used to reduce the chance that a damaging vulnerability is missed as a smart contract gets uploaded to the EVM. It has been noted in multiple places that many tools are incapable of accurately identifying certain vulnerability types. It is concerning to note that certain bugs like bad randomness were entirely missed by all eight benchmarked tools. If a tool existed to identify such errors, it would likely be highly desired for screening those identified vulnerabilities. However, since developing such a tool is beyond the scope of this project, the subject is laid to rest. Finally, the recommendation for the single best tool and best combination of tools is explained below.

4.1 Singular Tool Use

Conkas has been identified as the best solo tool to use when analyzing smart contracts. It has a solid detection rate: around 90% for the most common vulnerability types explored in this

study. It also generated, by a substantial margin, the fewest number of false negative signals for vulnerability detection. False negatives are the most damaging form of signal because of their tendency to erroneously inform the programmer that no vulnerability exists. Conkas scored passably in terms of the number of false positive signals—not substantially better or worse than other tools. Finally, the only significant drawback of using Conkas would be how slow it is on average. Being on average forty times slower than Slither, users will have to wait for quite a while to get a contract analyzed, particularly for long and complicated contracts. This tool might not be preferable for bulk-scanning the entire Ethereum mainnet. Instead, Slither would almost certainly be the best candidate. This is because in the time it took for Conkas to process each contract that exists on the EVM, Slither would have completed the analysis forty times over.

4.2 Multiple Tool Use

Conkas does not exist in isolation however. Slither, Mythril, and SmartCheck are also capable static analysis tools which perform well in most circumstances. If a programmer has the patience and ability to sift through the results of multiple tools, including the results from Slither, Mythril, and SmartCheck, it would certainly widen the range of bugs that potentially get caught. In specific instances, these tools outshine the capabilities of Conkas itself. For instance, Slither had an amazing record when finding reentrancy vulnerabilities. It generated only a single false negative among the curated smart contracts while raising fewer false positives than Conkas. Mythril and SmartCheck performed very similarly to Conkas in terms of discovering unchecked call bugs. Finally, SmartCheck could also be used to sift for access control violations, but the programmer should be prepared to identify the numerous false positives that the tool flags. On the subject of bulk-scanning the entire DeFi space, Slither and Mythril would be a strong duo to consider using on every contract in the EVM.

REFERENCES

[1] D. Perez and B. Livshits, "Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited", Proceedings of the 30th USENIX Security Symposium, 2021. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/perez

[2] A. Ghaleb and K. Pattabiraman, "How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection", Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020. Available: 10.1145/3395363.3397385

[3] Bo Jiang, Ye Liu, and W. K. Chan. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In Proc. International Conference on Automated Software Engineering, pages 259–269, 2018.

[4] Yu Feng, Emina Torlak, and Rastislav Bodík. 2019. Precise Attack Synthesis for Smart Contracts. CoRR abs/1902.06067 (2019). arXiv:1902.06067 http://arxiv.org/abs/1902.06067

[5] PeckShield, “New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018–10299),” Medium, May 02, 2018. https://tinyurl.com/yd78gpyt [Accessed: 04-Dec-2021].

[6] “What is DeFi?,” www.coinbase.com. https://www.coinbase.com/learn/crypto-basics/what-is-defi

[7] “Home,” ethereum.org. https://ethereum.org/

[8] Ferreria, P. Cruz, T. Durieux and R. Abreu, "SmartBugs: A Framework to Analyze Solidity Smart Contracts", Ieeexplore.ieee.org, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9285656. [Accessed: 02-Dec-2021].

[9] "Immunefi", Immunefi, 2021. [Online]. Available: https://immunefi.com/. [Accessed: 09-Dec-2021].

APPENDIX A: DURATION PLOTS OF ANALYSIS TOOLS

Pictured below are several variations of plots of the runtime of Ethereum smart contract static analysis tools. The various renditions are intended to make it easier to view the differences between certain tools. Figure A.1 is the raw duration plot of each analysis tool. This figure was also pictured as Figure 3.1 in the Results chapter.

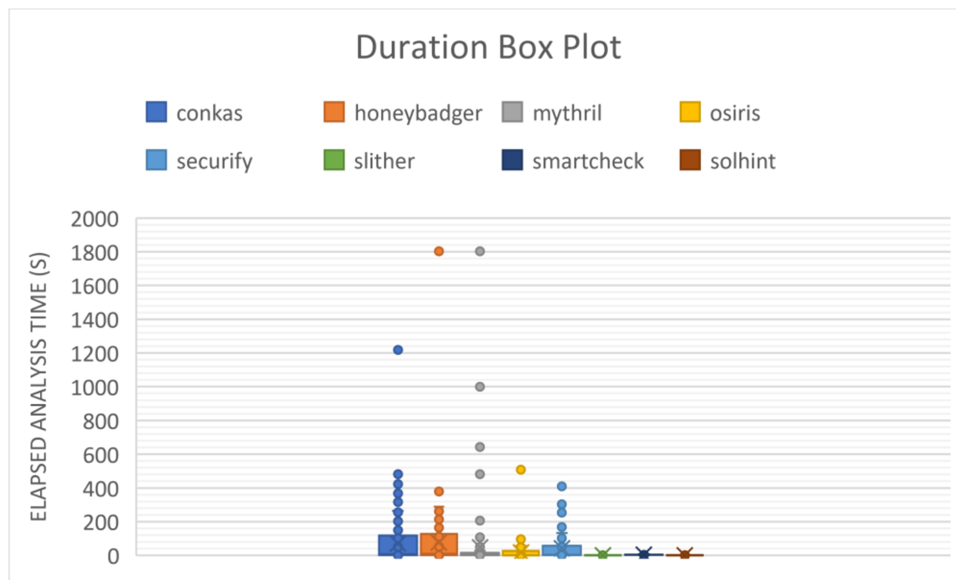

Figure A.2: A plot of the duration (in seconds) of each tool on each contract.

Above is a box plot of the duration of each tool on each curated contract in raw seconds. Pictured below in Figure A.2 is the same duration plot after being divided for each case by the file size of each contract. Like the previous plot, most contracts are analyzed quite quickly, with certain outlying exceptions.

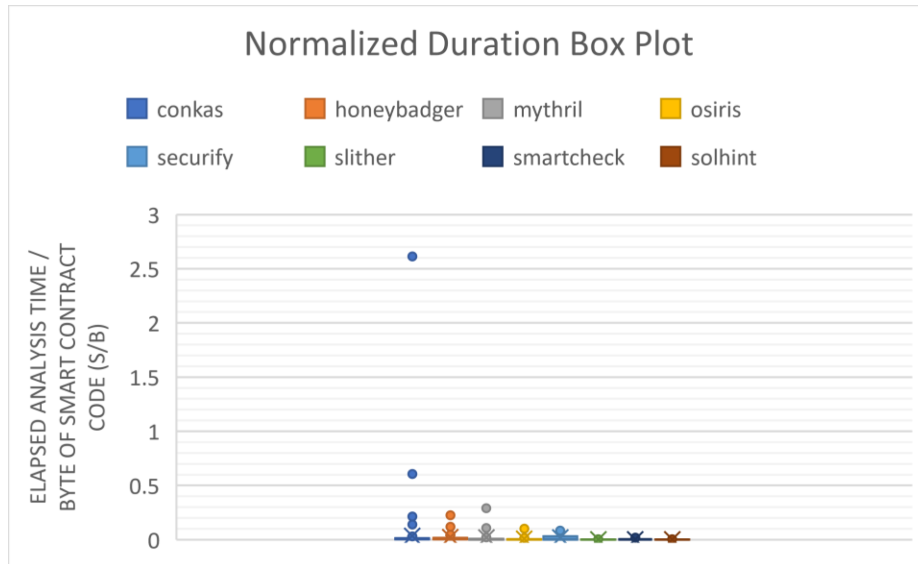

Figure A.3: Normalized duration box plotted as (seconds/byte) for each smart contract.

Next, to easily distinguish the actual differences between each analysis tool, the first graph is re-plotted on a log scale in Figure A.3. The average runtime of each analysis tool can be show to vary nearly as much as a full magnitude between certain tools.

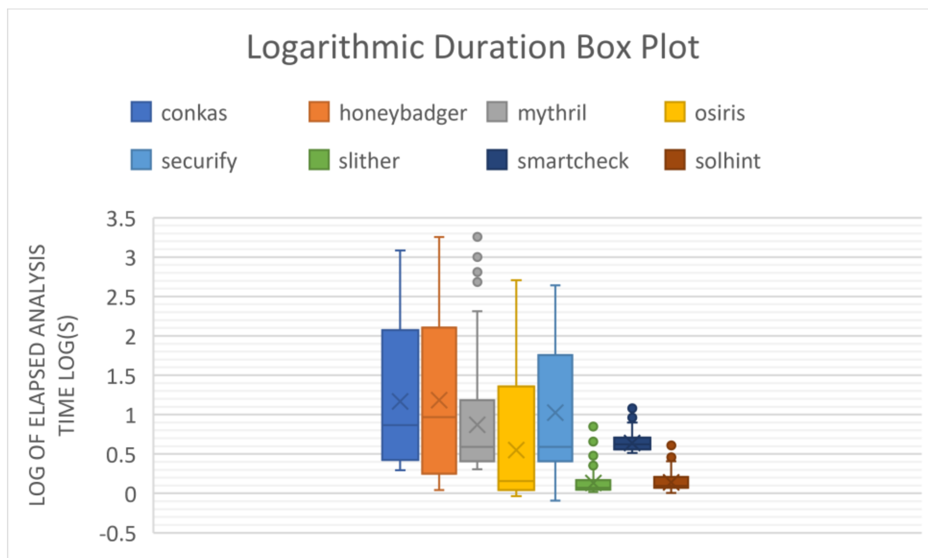

Figure A.3: A plot of the duration (in seconds) of each tool on each contract (in log scale).

Finally, the normalized duration from Figure A.2 is plotted in log scale to make the differences in duration ratios possible to distinguish. This is the same plot which has been

reproduced as Figure 3.2. In figure A.4, even after normalization of runtimes with file size, there are still huge variations in runtime with certain analysis tools on some smart contracts. Moreover, the differences in normalized runtime are still vast between analysis tools.

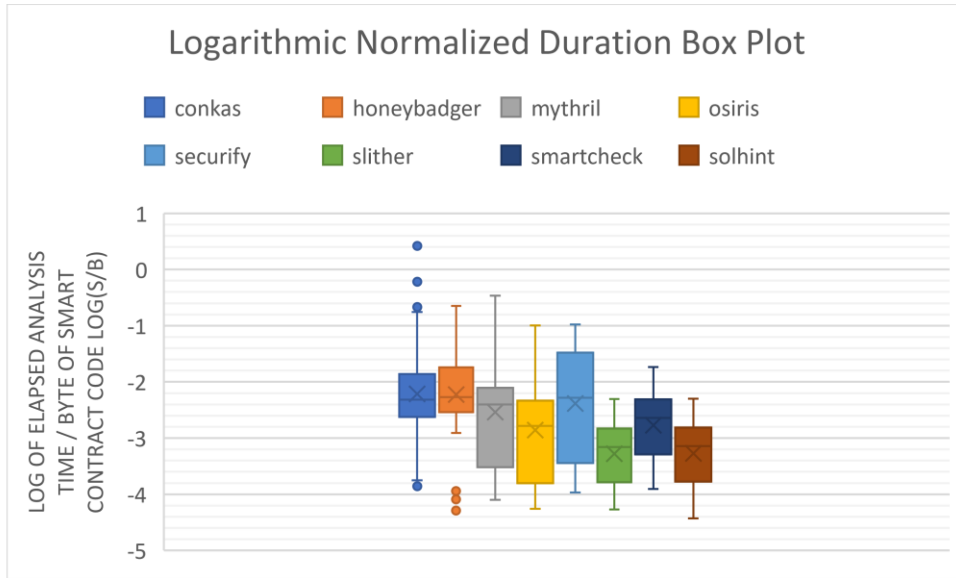

Figure A.4: Relative runtime performance of eight smart contract analysis tools normalized by file size (in log scale).

APPENDIX B: DURATION PLOTS OF ANALYSIS TOOLS WITH CERTAIN METRICS

To better picture the relationship between time taken for smart contract analysis versus certain metrics, four static analysis tools' performance on each individual contract have been graphed as a scatter plot. The three tools selected were Mythril, Osiris, and Securify. Each tool has a different amount of correlation between runtime and normalization metrics. On some, the correlation is very slight or highly noisy. Slither was chosen in Chapter 3 because of its easy-to-see correlation between runtime and a normalizing metric. Looking at Appendix A, it can be inferred that the tools with the smallest variance had the highest correlation between runtime and metrics. Pictured below in Figure B.1 is Mythril normalized by file size.

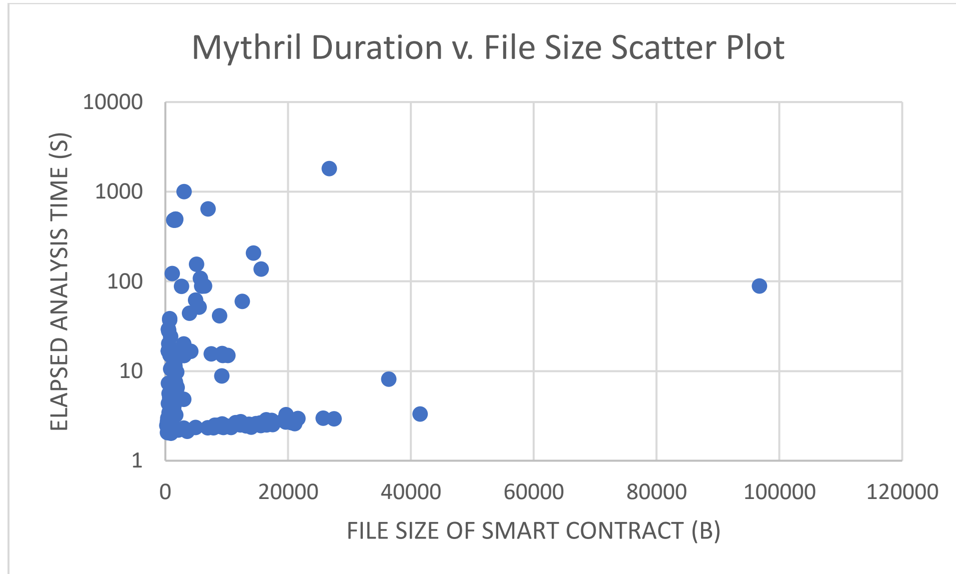

Figure B.1: Mythril runtime on each smart contract compared to file size.

Here, a clear dichotomy emerges. Some contracts appear to correlate with file size, while others outlie substantially. The most reasonable explanation for this is that certain analysis routines inside of Mythril take a much longer time to compute than others. Since some contracts trigger different routines, file size becomes less of a dominant parameter. A similar story is told in Figure B.2 with Osiris and lines of code.

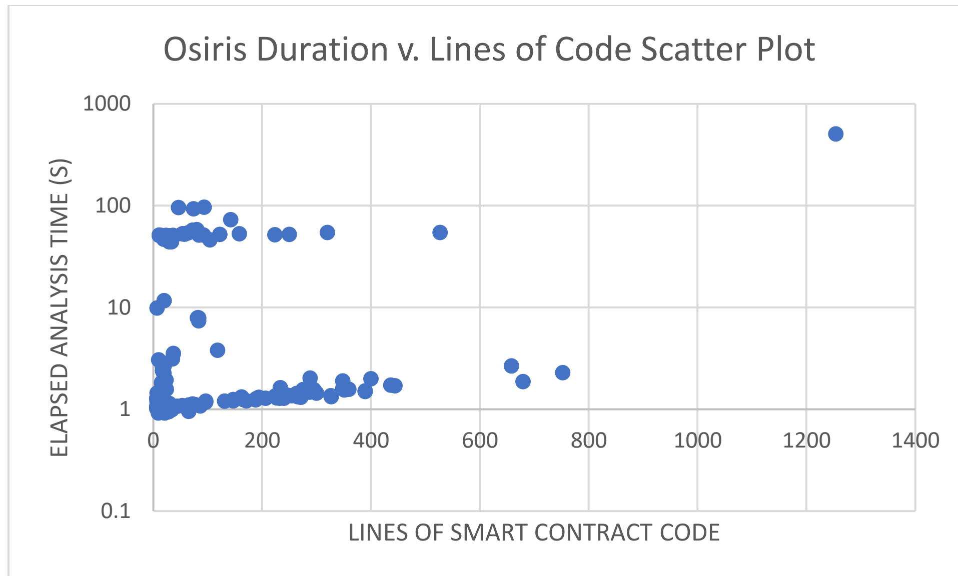

Figure B.2: Osiris runtime on each smart contract compared to lines of code.

Certain contracts appear to take a fixed amount of time, implying that there are features in some contracts which trigger less performant blocks of analysis inside of Osiris. Code complexity is the final metric used to compare the runtime of analysis tools on each smart contract. The total number of “function,” “event,” “constructor,” and “modifier” keywords were detected and summed up for each contract for Figure B.3.

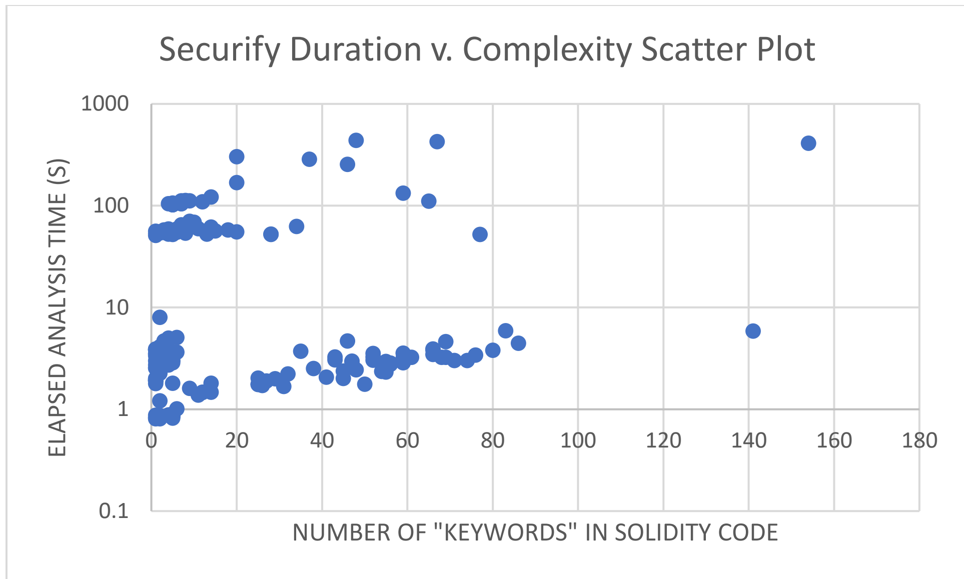

Figure B.3: Securify runtime on each smart contract compared to code complexity.