HOW EFFECTIVE CAN WE DETECT SOFTWARE VULNERABILITIES USING CODE CLONES? - A CASE STUDY ON ETHEREUM SMART CONTRACTS

YINGHANG MA

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL AND COMPUTER ENGINEERING
YORK UNIVERSITY
TORONTO, ONTARIO

JULY 2023

Abstract

Smart contracts are self-executing programs that are deployed on blockchain platforms to provide services and handle transactions. Smart contracts, which are usually implemented with the Solidity programming languages, exhibit different code characteristics (e.g., code complexity measures) compared to the software projects written in conventional programming languages(e.g., Java, C++, and C#). In addition, there is a much higher level of code-to-clone ratio for the Solidity contracts deployed on the Ethereum blockchain platform compared to conventional software projects. These differences can impose a wider spread of security risks, as there can be many more cloned code snippets with each reported vulnerability. These clones may suffer from the same security problems as their cloned counterpart. Hence, in this thesis, we have conducted an empirical study on the effectiveness of leveraging code detection techniques to identify software vulnerabilities in the Solidity contract code. To control the confounding factor of the configurations of the clone detection tools, we have experimented with a set of configuration tuning approaches, while keeping everything else constant. As a result, various configuration setups were suggested from these tuning approaches, and clone detection experiments were conducted under these configuration setups. The resulting reported clones from these experiments were then carefully analyzed qualitatively and quantitatively to: (1) assess the performance of these tools, and (2) to understand the rationales behind falsely reported clones and misreported clones. Our results showed that the configurations of the clone detection tools matter a lot while being applied to vulnerability detection. After carefully tuning these configurations, we showed that the tools tuned under the context-specific tuning approaches can achieve significant improvement while detecting vulnerable smart contracts (e.g., 44 times and 98.61% and improvement in F1-scores for SourcererCC and NiCad, respectively) compared to the start-of-the-art configuration tuning approach (a.k.a., general agreement score-based approach). As a by-product of this research, we have also found 330 vulnerable smart contracts which were previously unreported. The results reported in this thesis highlighted the need for further research into context-specific clone detection and management. This thesis will be useful for software engineering and security engineering researchers and practitioners in the domain of blockchain-based applications.

Acknowledgements

I would like to express my deepest gratitude to the following individuals and organizations who have played a significant role in the completion of this research:

First and foremost, I would like to sincerely thank my supervisor, Dr. Zhen Ming (Jack) Jiang, for his invaluable guidance, expertise, and continuous support throughout my M.A.Sc project. His insightful feedback and encouragement have been instrumental in shaping the direction of this research.

I would also like to thank the members of my research committee, Dr. Song Wang and Dr. Manar Jammal, for their valuable insights and suggestions, which greatly enhanced the quality of this study.

Finally, I want to express my deepest gratitude to my lab mates, family, and friends for their unwavering support, understanding, and encouragement throughout this journey.

Table of Contents

Abstract **ii**

Acknowledgements **iii**

Table of Contents **iv**

List of Tables **vi**

List of Figures **vii**

1 Introduction **1**
1.1 Thesis Organization 3

2 Background and Related Work **4**
2.1 Smart Contracts and their Associated Technology 4
2.1.1 Terminology 4
2.1.2 Empirical Studies on Ethereum Smart Contracts 5
2.1.3 Code Cloning 7

3 Preliminary Study **9**
3.1 Data Collection 9
3.2 Clone Detection Tools 10
3.3 Data Analysis 11
3.4 Discussion and Case Study Setup 12

4 RQ1 (Tool Comparison) **16**
4.1 Data Collection 16
4.2 Experimentation 17
4.2.1 Genetic Algorithm Results 19
4.3 Data Analysis 19
4.3.1 Quantitative Analysis 20
4.3.2 Qualitative Analysis 20

5 RQ2 (Effectiveness Assessment) 25
5.1 Data Collection . . . 25
5.2 Experimentation . . . 25
5.3 Data Analysis . . . 25
5.3.1 Quantitative Analysis . . . 26
5.3.2 Qualitative Analysis . . . 28

6 RQ3 (Context-specific Optimization) 31
6.1 Data Collection . . . 31
6.2 Experimentation . . . 31
6.3 Result Analysis . . . 33
6.3.1 Quantitative Analysis . . . 33
6.3.2 Qualitative Analysis . . . 33

7 Discussion and Future Work 39
7.1 Other Clone Detection Tools . . . 39
7.2 Clone Management in the Context of Solidity Contract Development . . . 40
7.3 Vulnerability Discovery and Reporting . . . 41
7.4 Similarity Measurement . . . 42

8 Threats to Validity 43
8.1 External Validity . . . 43
8.2 Internal Validity . . . 43
8.3 Construct Validity . . . 44

9 Conclusions 45

List of Tables

3.1 Breakdown of clones types among the reported vulnerable functions 12

4.1 Optimal configuration outputted by the Genetic Algorithm under the GAS-based approach 19

5.1 Evaluation results for the GAS-based configuration setup based on the GA range defined in [98]. 28

6.1 The resulting optimal configurations after different GA tuning . GAS, O_u, O_i, O_s refer to the GA process while using the General-agreement-score/Union/Intersection/Separately tuned as the objective function, respectively. 33

List of Figures

3.1 Our process for the preliminary study 9
3.2 Venn diagram for clone pairs reported by NiCad and SourcererCC 11
3.3 Histograms for the cluster size in Union and Intersection datasets 14
3.4 An Example of the Missing Clones from NiCad under the Default Configuration 15

4.1 Agreement score in each generation for GA algorithm under GAS-based setup. 19
4.2 Venn Diagram for the Clone Pair Analysis for NiCad and SourcererCC. . . . 20
4.3 Examples of unique clone pairs outputted from NiCad and SourcererCC. . . 21
4.4 Pie chart showing reasons behind the unique clone pairs under default and GAS-based setup. 23

5.1 Clone pair samples drawn from the vulnerable clone pairs. 27
5.2 Stack Bar Chart Analysis 29
5.3 Clone type analysis under the GAS-based configuration setup 29
5.4 Pie chart showing reasons behind the misreported vulnerable clone pairs under the GAS-based setup. 30

6.1 Precision, Recall, and F1-score under different configuration setups as a result of various GA tuning processes. ('default' stands for default configuration) . 34
6.2 O_u-based result 35
6.3 Stack bar chart analysis for the positive and the negative cases of `Type-1, 2, 3` clones. 35
6.4 Clone types of positive cases under the optimal configuration setup (O_u) for NiCad and SourcererCC. 36
6.5 Venn diagram for reported recall pairs between NiCad and SourcererCC under O_u-based configuration setup. 36
6.6 Misreported reason analysis for the O_u-based configuration setup. 37
6.7 Sample clone pairs written in Solidity and Java. 38

7.1 Public Function Example 40
7.2 Internal Function Example 41

Chapter 1

Introduction

Blockchain technology presents a revolutionary approach to storing information in a distributed manner without alternation [91]. Ethereum is a large-scale, decentralized, public blockchain platform that enables developers to deploy and execute smart contracts [31]. Smart contracts refer to autonomous contracts that are coded to facilitate transactions on the blockchain platform [71]. Smart contracts have been used for various purposes like ICOs (Initial Coin Offers), crowdfunding initiatives, or as a component of decentralized applications (ĐApps). Solidity is generally the programming language used to develop smart contracts [31].

Code clones are snippets of duplicated code as a result of copy-and-paste activities from developers [50]. There are pros and cons associated with code cloning. On one hand, clones can be created intentionally as a natural way to code by using code templates [50, 53] or to mitigate risks [29]. On the other hand, without careful clone management [105], clones can be harmful as they can introduce bugs [63] or even security vulnerabilities [46]. Although there are many security vulnerability issues reported for Ethereum smart contracts (e.g., [67, 43, 26, 40]) in the past, there are no studies focusing on examining the relationship between code cloning and software vulnerability in the context of Solidity smart contracts. This is mainly due to the following four challenges:

1. **Code Characteristics**: Existing empirical studies showed that compared to conventional programming languages like Java, C++, and C#, Solidity smart contract code exhibited different characteristics like code complexity [75] and code-to-clone ratios [60]. There were much more clones (about 79.2%) among the smart contracts deployed on the Ethereum blockchain platform [60]) compared to conventional software projects. Hence, when software vulnerabilities are spotted in the context of Solidity smart contracts, it is vital to detect and mitigate the risks from their clones in order to minimize the impact of such issues.

2. **Tool Support**: There are many clone detection approaches [88, 82, 79, 84, 48, 55, 66]. Unfortunately, many of these tools either did not support the Solidity programming language (e.g., CCFinderX [79]), or did not support Solidity function-level clone

detection (e.g., SourcererCC [84]), or failed on the latest version (e.g., 0.8 and above) of the Solidity contract code (e.g., Deckard [48], NiCad [55], and Birthmark [66]).

3. **Tool Configurations**: As shown in the previous studies, the performance of the clone detection results was significantly impacted by the configurations of clone detection tools [98]. To make matters worse, such configurations may need to be tuned depending on the actual use cases, such as detecting clones in the testing code [96]) or detecting IR (Intermediate Representation) level clones [100]. Unfortunately, there is no work dedicated to providing configuration tuning guidance specifically on vulnerability detection.

4. **Evaluation**: There are many clone evaluation studies focusing on the effectiveness of detecting clones in general [98, 81, 39, 90]. However, different clones are used and treated differently depending on their usage context. For example, not all clones can be refactored [17, 93]. Similarly, not all clones will lead to bugs [45] or software vulnerabilities [59]. Unfortunately, there are no existing studies that thoroughly evaluate the outputs from various clone detection experiments in the context of vulnerability detection.

In this thesis, we have conducted a large-scale empirical study on detecting software vulnerabilities using code clones in the context of Ethereum smart contacts. To cope with the above challenges, we first extended the capabilities of two popular state-of-the-art clone detection tools: (1) an AST-based clone detection tool, NiCad [80], and (2) a token-based tool, SourcererCC [84], to support the cloning analysis on the Solidity contract code. Then we experimented with a range of different tool configurations for both tools while detecting clones on the Solidity contract code. To thoroughly evaluate various clone configuration setups, we have carefully constructed a high-quality evaluation dataset for vulnerability detection in the context of Ethereum smart contracts. This dataset consisted of 478 Solidity contract files from real-world CVE (Common Vulnerabilities and Exposures) issues from 2018 to 2022. We intentionally removed certain reported vulnerable issues that either missed the reproducibility steps or cannot be successfully reproduced and verified by us. To ensure the correctness of the detected clone functions in our results, we have verified over one thousand outputted clones that are potentially vulnerable by manually executing and verifying the reproducibility steps mentioned in these issue reports. This corresponds to over 300 hours of one person's work. As a result, we have verified 330 additional clones which exhibit the same problems but are not reported as CVEs or GitHub issues. While cross-examining the clones among different configurations, we have also conducted both quantitative studies and qualitative studies to assess the tool performance and to explain the rationals/causes behind these differences. Our findings highlight the pros and cons of various clone detection techniques and provide practical guidance on configuration tuning in the context of vulnerability detection on Ethereum smart contracts. The contributions of this thesis are:

1. This is a comprehensive study that thoroughly examined the relations between code cloning and software vulnerabilities in the context of Ethereum smart contacts. Our

results showed that the majority (91.30%) of the reported vulnerability issues have clones, which could be effectively (up to 0.99/0.85/0.89 in precision/recall/F1-score) detected by existing state-of-the-art clone detection techniques (e.g., NiCad and SourcererCC) when we carefully tuned their tool configurations.

2. Although code clones are identical or very similar code snippets, different clones are needed in different use cases (e.g., vulnerability detection, refactoring, and change propagation) or different programming languages (e.g., Java, C++, or Solidity). Existing approaches to evaluating clone detection techniques (e.g., [98, 100]), only tuned the tool configurations based on the amount of overlap (a.k.a., General Agreement Score (GAS)-based approach) of the resulting code clones outputted by different tool setups. Our case studies showed that depending on the tools our proposed *Context Specific (CS)*-based tuning approach was significantly better (e.g., 98.61% better for NiCad and over 44 times better for SourcererCC in F1-score) than the GAS-based approach. As clones are now being used in practice under various contexts (a.k.a., use cases, and programming languages), the results in this thesis called for more research on context-specific clone detection and evaluation research in the future.

3. To facilitate replication and further research in this area, we provided our replication package [102] which contains our updated tools, the curated datasets, and reported additional vulnerable functions. The updated version of NiCAD and SourcererCC can perform clone detection on the latest version(0.8 and above) of the Solidity contract code. In addition, we have also reported our discovered 330 additional vulnerable clones to the open-source community.

1.1 Thesis Organization

Our thesis is organized as follows: Chapter 2, we introduce the background and the related works of our study. Chapter 3 describes our preliminary studies and provides motivations for our three Research Questions (RQs). Chapters 4, 5, and 6 presents the details of our three research questions. Chapter 7 discusses several topics related to the results of our RQs and presents some future work. Chapter 8 explains the threats related to this thesis and Chapter 9 concludes this thesis.

Chapter 2

Background and Related Work

In this chapter, we provided two areas of necessary background and prior works related to this thesis: (1) smart contracts and their associated technology, and (2) code cloning.

2.1 Smart Contracts and their Associated Technology

We first presented terminologies related to smart contracts. Then we discussed various empirical studies conducted in this area.

2.1.1 Terminology

Blockchain is a decentralized and distributed ledger that is immutable. It enables multiple parties to maintain a shared, transparent, and secure record of transactions without relying on a central authority [91]. Blockchain is recognized as a decentralized database that is distributed across a network of computers and lacks a central authority. It is comprised of interconnected blocks, with each block containing multiple transaction records. The blocks are linked to their preceding block, thereby creating a chain-like structure, which is why it is referred to as a 'blockchain'.

Cryptocurrencies are digital or virtual currencies that use cryptography for security and run on decentralized networks such as blockchains. Cryptocurrencies are intended to be secure, open, and impervious to fraud. Bitcoin [23] is a decentralized digital currency and a groundbreaking concept in the realm of finance and technology. The cryptocurrency used in the Ethereum network is called Ether (ETH). Ether serves as the native currency and fuel for the Ethereum platform, which is a decentralized blockchain-based platform that enables the execution of smart contracts and the development of decentralized applications (DApps).

Ethereum is a public blockchain platform that operates in a decentralized manner. It was established in 2015 by Vitalik Buterin [31]. The platform is open-source in nature and facilitates the development of decentralized applications (DApps) and the execution of smart contracts by software developers. In contrast to Bitcoin [23], Ethereum was conceptualized

as a platform with the capacity to construct DApps and perform smart contract operations instead of serving as a digital currency.

Smart contract is a computer program that is designed for general-purpose use. The Solidity programming language is commonly utilized for scripting smart contracts in the Ethereum platform. Solidity is an object-oriented programming language that exhibits a syntax that bears a likeness to that of Java. The organization of the source code for a Solidity smart contract is structured around subcontracts, interfaces, and libraries. Smart contracts are deployed on the Ethereum blockchain platform in bytecode format. By searching the address of the smart contract on the Etherscan website [36], you can verify the source code with their corresponding bytecode (a.k.a., verified smart contracts). Etherscan website also provides traceable source code and transaction history for verified smart contracts.

2.1.2 Empirical Studies on Ethereum Smart Contracts

There are a few works, which study the phenomena of code cloning in the context of smart contracts.

Kondo et al. [60] conducted a large-scale empirical study on the verified Ethereum smart contracts to investigate the prevalence and impact of code cloning. They found 79.2% of the smart contracts are clones of each other. When clustering clones together, 9 of the top 10 clone clusters are token managers and the cloned contracts within the same clusters usually come from different authors. In addition, they also reported code similarities between the studied smart contracts and the library functions offered by the OpenZeppelin project, which provides utility functions for writing secure and efficient smart contracts.

Faizan et al. [55] replicated the same study of [60] with a focus on function-level cloning. They confirmed that Kondo et al.'s findings also hold at the function level.

Pierro and Tonelli [55] studied the code cloning between existing verified Ethereum smart contracts and the OpenZeppelin code. They found that certain code clones can be resolved by reusing the functions provided inside the OpenZeppelin project.

Our thesis is different from the above works, as we focused on studying the relationship between code clones and software vulnerabilities in the context of Ethereum smart contracts. In addition, the above works regarding Solidity clone detection only used one clone detection technique under one configuration setup or their own configuration setups without tuning. We experimented with two clone detection techniques (NiCad and SourcerCC) over a range of different tool configurations, as previous studies show that different clone detection techniques can detect different types of clones [89] and tool configurations can significantly impact the quality of the outputted clones [98]. Public blockchain platforms like Ethereum handles millions of transactions every day. Hence, security is essential for such a platform. Several studies were done to categorize the vulnerabilities reported in such platforms.

Yi et al. [101] conducted an empirical study on the security vulnerabilities by going through the GitHub commits, issues, and pull requests from four types of popular blockchain systems: Bitcoin, Ethereum, Monero, and Stellar. They have found that 70% vulnerabilities are vulnerabilities similar to traditional software systems, whereas the remaining ones are

more domain specific.

Soud et al. [87] conducted a similar study by analyzing the vulnerabilities from GitHub, Stack Overflow, and popular vulnerability databases (i.e., National Vulnerability Database and Smart Contract Weakness Registry). They leveraged a card-sorting approach to systematically categorize these issues.

To mitigate such security risks, various techniques (e.g., fuzz testing, formal verification, symbolic execution, static analysis, and clone detection [104, 16]) have been experimented with to detect vulnerabilities in the Ethereum platform. Although the authors did not experiment with any of the clone detection tools, they did note in the paper [104] that they believe the vulnerability detection performance can be further improved if code clones were used.

Our work complements [104, 16], as we have carefully evaluated the effectiveness of the clone detection technique in the context of vulnerability detection by manually checking and reproducing these issues.

In addition to code cloning and security vulnerabilities, researchers also studied other artifacts of Ethereum smart contracts.

Oliva et al. [75] conducted the first exploratory study on the verified smart contracts deployed in the Ethereum blockchain platform. They have studied the activity levels of different smart contracts and their associated use cases. They also reported different characteristics while comparing the code complexity measures of smart contract code with source code from traditional software applications written in Java, C++, and C#.

Jiang et al. [49] investigated the calling relation among the Ethereum smart contracts and characterized them based on different types of user indices and contract indices.

Liao et al. [65] conducted a large-scale empirical study on the assembly code inlined in the Ethereum smart contracts. They have found that code inlining is quite prevalent in Ethereum smart contracts. The rationales of code inlining can vary, such as producing gas-efficient bytecode and easing smart contract development.

The execution of the smart contracts is carried out on miner's machines. The amount of computation resources is called *gas usage*. Miners are paid by the transaction fees, which are the product of the amount of gas usage to verify such contracts and the specified gas prices in those contracts. Zarir et al. [103] characterized the gas consumption of different Ethereum smart contracts. They have found that most miners prioritize smart contract transactions only based on the gas price and one-quarter of the functions which are frequently executed ($ge10$) exhibit unstable gas usage patterns.

Vacca et al. [94] studied the relation between the readability of smart contract codes and their associated gas consumption patterns. They have found smart contract code exhibit lower readability than traditional software code. Some readability metrics significantly correlate with their gas usage.

Sorbo et al. [34] identified a set of code smells and static code metrics which could potentially lead to high gas consumption costs. Brandstätter et al. [22] proposed a set of rules to optimize the gas consumption of smart contract uses based on various code optimization strategies such as loop, space-for-time, etc.

Pacheco et al. [76] conducted an empirical study on the transaction processing time in the Ethereum platform. They have found that most of the transactions can be processed within eight minutes and transactions with higher gas prices tend to be processed faster. They have also developed a statistical model to estimate the transaction processing time, which outperforms the state-of-the-art existing services.

The proxy pattern is a common design pattern to facilitate future code changes and software updates. Ebrahimi et al. [35] studied the use of proxy patterns amongst the Ethereum smart contracts and reported an increasing upward trend in the adoption of this pattern over time.

Xi and Pattabiraman [99] reported wide use of low-level functions, which are discouraged by the more recent releases of the Solidity language. To mitigate this problem, they also developed a tool to automatically replace these functional calls with their secure alternatives.

2.1.3 Code Cloning

Code cloning is a prevalent software development methodology that involves copying and pasting code fragments within a project or across multiple projects [50]Code clones can be further categorized into the following three types [18, 81]:

1. `Type-1`: The code fragments are identical except for differences in white spaces, layout, and comments.

2. `Type-2`: The code fragments are syntactically or structurally identical, with the exception of differences in identifiers, literals, types, layout, and comments.

3. `Type-3`: The code fragments are replicated with alterations such as modified, appended, or deleted statements, along with differences in variables, values, spacing, formatting, and annotations.

4. `Type-4`: The code fragments are implemented with differently but share the same functionality.

Various techniques have been proposed to automatically detect clones, such as text-based, token-based, metric-based, AST-based, and graph-based techniques [33, 62, 61, 70, 73, 83, 32, 51, 77]. In addition, researchers have extended these techniques so that clone detection can not only operate on source code data, but also on other artifacts such as IR [86, 100, 24], bytecode [66], and binaries [42]. Various studies have been carried out to empirically evaluate the effectiveness of clone detection techniques [18, 81, 100]. All the above works focused on evaluating clone detection on traditional software systems written in programming languages like C/C++ in a general context, whereas our work evaluated the effectiveness of clone detection on Solidity smart contracts in the context of vulnerability detection. In addition, different from [98] and [100], which tuned the tool configurations mainly based on the amount of agreement amongst different tools and evaluates them in a general context, our work

evaluated the effectiveness of clone detection under a range of different tuning approaches specifically in the context of vulnerability detection.

On one hand, code clones can be harmful to software quality as they can introduce bugs [63, 44, 72, 19]. On the other hand, clones can be created and maintained intentionally [57] as a way to develop features (e.g., code templating [53, 27]) or to minimize risks [50, 29]. Some clones can be created accidentally or unintentionally due to protocol specifications [52] or language limitations [17]. Clones can be used for experimental purposes or better program comprehension [56]. Clones have also been applied to detect software vulnerabilities (e.g., [59, 58]).

Our work is different from [66] as we focused on analyzing the source code instead of the Ethereum bytecode. We have also discussed various limitations and future work associated with leveraging bytecode level clone detection for detecting Ethereum smart contract vulnerabilities in the Discussion chapter 7.

VOLCANO [85] is the first work, which leveraged clone detection to detect vulnerabilities in the Ethereum smart contracts and studies their evolution. However, they only focused on one configuration setup under one tool (NiCad) and they did not evaluate thoroughly the performance of the resulting clones. Our thesis addresses both of these issues.

Chapter 3

Preliminary Study

In this chapter, we conducted a preliminary study on the degree of cloning among the reported vulnerable functions. Figure 3.1 showed the process of our preliminary study. We first collected a set of vulnerable Ethereum smart contract functions from CVE reports and GitHub sites. Then we conducted clone analysis among these functions using clone detection tools and analyzed the outputs from these two tools. The analysis results from this preliminary study motivated us to further examine the relations between code clones and software vulnerabilities in depth in the subsequent chapters.

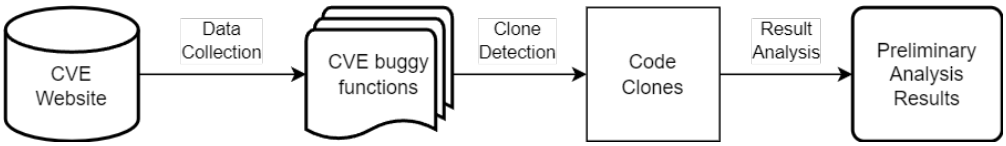


Figure 3.1: Our process for the preliminary study

3.1 Data Collection

Common Vulnerabilities and Exposures (CVE) [97] is a public website, which documents security vulnerabilities, exposures, and links to detailed reports. The goal of CVE is to create a system for facilitating the dissemination and interchange of security vulnerability information among security experts and entities. On January 18th, 2023, we searched the CVE website [97] using the keywords like "Ethereum", "function", and "Solidity" and found 516 CVE reports as a result. Then we removed CVE reports that are not related to the Solidity contract (e.g.,`CVE-2022-35936` [7]), written in another programming language (e.g., Vyper code issues like `CVE-2022-24788` [6]), or caused by misbehavior from the users (e.g., `CVE-2018-14085` [3]). To ensure reproducibility, we performed another round of manual filtering to filter out the cases which do not contain reproducibility steps or whose issues can not be accurately reproduced by us even if the same process described in their CVE reports is repeated. For example, `CVE-2020-20178` [5] and `CVE-2018-18425` [4] were removed as we

either cannot find their reproducibility steps or cannot reproduce the problems by following their steps. We also moved out vulnerable functions that do not have a CVE ID.

As a result, 38 cases were removed and 478 CVE cases remained. One CVE case corresponds to one CVE report, which has a unique CVE ID. Take `CVE-2018-12959` [2] for an example. This CVE report has a unique ID: `CVE-2018-12959`. It corresponds to one vulnerability, which is related to a unique contract address deployed in the Ethereum platform for this smart contract.

In CVE report website [97], the issue reporter provided a URL (e.g., `CVE-2018-12959` [2]), pointing to another website (e.g., GitHub), which contains more details regarding this issue. The linked GitHub report could contain multiple vulnerability issues, which may suffer from the same type of vulnerability. These vulnerable functions can be clones of each other, or simply the same code but are deployed in a different address. Similar to conventional software projects, although some of these reported bugs in GitHub do not have CVE IDs, they can also be vulnerabilities [78, 20]. But to ensure the quality of our dataset, we took the conservative approach and only kept the vulnerable cases that have corresponding CVEs signed to them. Among 478 CVE cases, 473 cases had vulnerabilities within just one function. For the remaining five cases, the vulnerability problems were embedded in two functions. We manually downloaded these smart contract codes and recorded the information about their vulnerable functions.As a result, we named these 478 vulnerable cases as dataset $Code_{CVE}$. $Code_{CVE}$ contained the vulnerable smart contracts along with their associated metadata including the website addresses for the linked detailed reports, the related contract names and their addresses, the contract deployment transaction hashes, and their contract block numbers in the Ethereum platform. By skimming through $Code_{CVE}$, we found many function-level clones inside. This observation motivates us to further examine the degree of code cloning from these reported vulnerable Ethereum smart contracts.

3.2 Clone Detection Tools

There are many existing works that leverage token-based [14, 64, 74] and AST-based [74, 46, 54] clone detection tools techniques to discover software vulnerabilities. Hence, in this thesis, we chose two state-of-the-art clone detection tools from these two approaches: NiCad [80] and SourcererCC [84] as our clone tools.

Although an extended version of NiCad was created to detect clones in the Ethereum smart contract before [55], this version of the tool was used to parse older versions of the Solidity code (version 0.4 to version 0.7) and failed on the latest version (a.k.a., version 0.8 and above). Hence, we updated the TXL [30, 28] Solidity grammar files for NiCad in order to parse Solidity grammar version 0.8 and above of the Solidity source code. The existing version SourcererCC only supports file-level Solidity clone detection since it does not provide scripts to extract the functional entities in Solidity contract files. Hence, we updated the SourcererCC tokenizer by leveraging Exuberant Ctags [47], which can report the exact line numbers for each function inside the Solidity contract to tokenize functions into tokens. The tokenized output is generated with two factors: a config file containing definitions for Solidity

separators and comments, and a SourcererCC tokenizing algorithm. With these factors, the tokenized functions can be processed by the subsequent similarity-matching steps inside SourcererCC. The patches for NiCad updates are now accepted by the tool repository, and SourcererCC patches are available for testing [69, 68] to facilitate replication and further research.

We performed clone detection tools using NiCad and SourcererCC on the above dataset under their default tool configurations. Both tools outputted a set of clone pairs. Then we used the Solidity AST parser [1] and Gumtree [38] to further classify the clone types (`Type-1, 2, and 3`) for these clone pairs.

3.3 Data Analysis

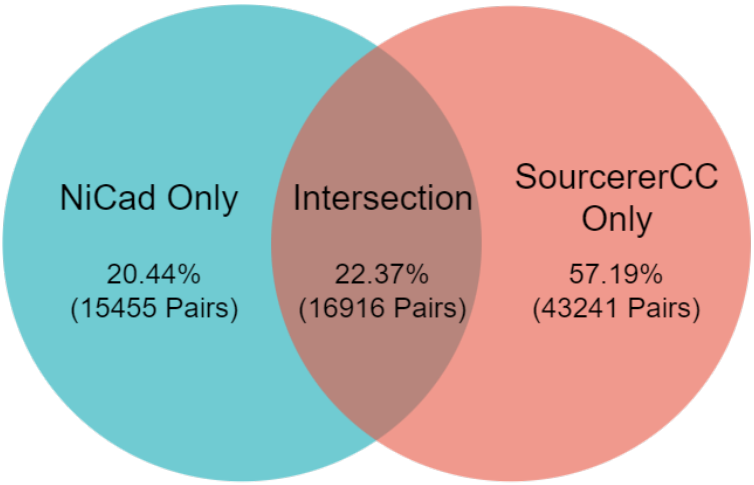


Figure 3.2: Venn diagram for clone pairs reported by NiCad and SourcererCC

Result Comparison We combined the results from NiCad and SourcererCC and removed the duplicates from them. This resulted in a total of 75612 clone pairs. Figure 3.2 showed the resulting Venn Diagram. (22.37%(16916) of the clone pairs were reported by both tools. SourcererCC reported (57.19% unique clone pairs, which took the largest part in the combined result. NiCad reported much fewer (20.44%) unique clone pairs than SourcererCC. Since results from NiCad and SourcererCC were different, we grouped the clone pairs from these two tools using Union ($Clone^{u}_{CVE}$) and Intersection ($Clone^{i}_{CVE}$) to represent the pessimistic and optimistic cloning scenarios. Table 3.1 showed the types of clones. The distribution of the clone types was quite different when considering the union or the intersection of the cloned results.In $Clone^{i}_{CVE}$, the majority (57.41%) were `Type-1` clones, whereas in $Clone^{u}_{CVE}$, the majority (56.15%) were `Type-2` clones.

Clusters of Clones In order to understand the spread of cloning, we clustered both the Union and the Intersection of the clone pairs. The results were visualized in Figure 3.3. In the Union result, the largest cluster had 361 vulnerable cloned functions. This accounted

Dataset X_c	Union	Intersection
clone pair Count	75612	16916
`Type-1`	36.96%	57.41%
`Type-2`	56.15%	38.95%
`Type-3`	6.89%	3.65%

Table 3.1: Breakdown of clones types among the reported vulnerable functions

for 74.74% of the total vulnerable functions. The second largest cluster size had 33 cloned functions, which accounted for 6.83% of the total vulnerable functions. In the Intersection result, the cluster size was smaller with the largest cluster containing 254 vulnerable functions (52.59% of the total reported vulnerable functions) and the second largest cluster size containing 52 vulnerable functions (10.77% of the total reported vulnerable functions).

Manual Sampling We manually examined a few clones to understand why NiCad reported much fewer clones than SourcererCC. The issue was similar to [95], in which the clone configuration needed to be tuned in order to perform better in different usage contexts (e.g., detecting clones in the testing code). If we lower the threshold of similarity from 80% to 60% in NiCad, many of the unique cloned from SourcererCC shown in Figure 3.4 can be detected by NiCad.

3.4 Discussion and Case Study Setup

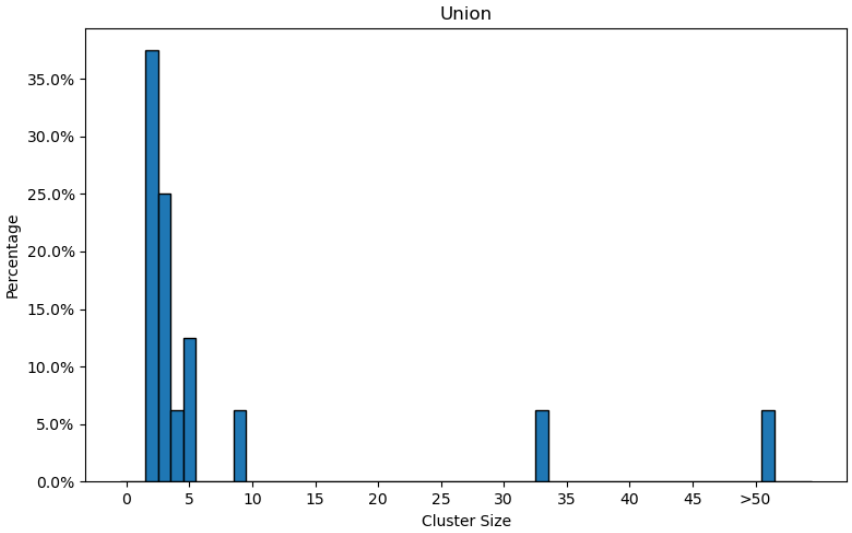
In summary, 91.30% of reported vulnerable functions were clones of each other. The cloning outputted from SourcererCC and NiCad had different clone pairs in their results. The configurations of these tools played a big difference in the resulting clones. Hence, these findings motivated us to do further investigations on the relationship between code clones and software vulnerabilities in the subsequent chapters 4 5 6. In particular, we wanted to answer the following research questions:

- **RQ1 (Tool Comparison): What is the performance of these clone detection tools while analyzing Solidity contracts?** As shown in the previous study [98], tool configuration is one of the confounding factors when comparing the results of different clone detection tools. Hence, in this RQ, we follow the same process as in [98] for tuning the configurations of the two studied clone detection tools in the context of the Ethereum Solidity contract code.

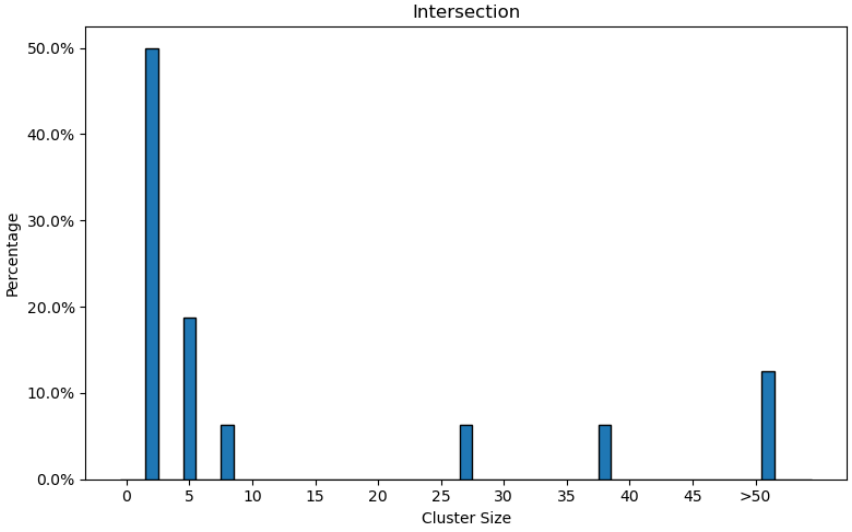
- **RQ2 (Effectiveness Assessment): Can we use the above tool configurations to evaluate the effectiveness of these clone detection tools for detecting CVE bugs for Solidity contracts?** Not all the reported clones would lead to software vulnerabilities. Hence, in RQ2 we wanted to use the tuned tool configurations from RQ1 and apply them for software vulnerability detection (a.k.a., flagging vulnerable contracts from reported CVE issues).

- **RQ3 (Context-specific Optimization): How effective are these clone detection tools for detecting Ethereum CVE vulnerabilities if we tune their configurations specifically for this use case?** In RQ2, we only tuned the tool configurations based on the degree of agreement between the tool outputs. In this RQ, we wanted to check whether the effectiveness of vulnerability detection would be further improved if we tune the tool configurations not only for the agreement but also specifically for this use case.

Each of these RQ was further divided into four parts: data collection, experimentation, data analysis, and result discussion and implications.



(a) Cluster size in Union dataset



(b) Cluster size in Intersection dataset

Figure 3.3: Histograms for the cluster size in Union and Intersection datasets

Reason	Clone Fragment 1	Clone Fragment 2

Reason: Does not reach NiCAD Smilarity threshold
Clone is in SourcererCC only
	function mintToken(address target,
	uint256 mintedAmount) onlyOwner public {
	balanceOf[target] += mintedAmount;
	totalSupply += mintedAmount;
	emit Transfer(0, this, mintedAmount);
	emit Transfer(this, target, mintedAmount);
	}
	CVE ID: CVE-2018-13173
		function mintToken(address target,
		uint256 mintedAmount) onlyOwner public {
		balanceOf[target] += mintedAmount;
		totalSupply += mintedAmount;
		Transfer(0, this, mintedAmount);
		Transfer(this, target, mintedAmount);
		}
		CVE ID: CVE-2018-13763

Figure 3.4: An Example of the Missing Clones from NiCad under the Default Configuration

Chapter 4

RQ1 (Tool Comparison)

The goal of this RQ was to compare the number of code clones from different clone detection tools in the context of Ethereum smart contracts. As noted in [98], the performance of clone detection tools varied significantly depending on their tool configurations. Therefore, we needed to control this confounding factor while conducting our comparison. Hence, in this RQ, we first described our process on how to collect the verified contracts from Ethereum. Then we tuned the configuration of NiCad and SourcererCC by following closely the approach presented in [98]. Subsequently, the tuned configurations for both tools were used to perform clone detection on the downloaded Solidity source code and the reported vulnerable functions. Finally, we conducted quantitative and qualitative analyses on the cloning outputs from these two tools.

4.1 Data Collection

Since we have already collected the vulnerable Solidity contracts in the preliminary study chapter (Chapter 3), we just needed to download all the necessary source codes for the verified Ethereum smart contracts in this RQ.

`Etherscan` [36] is a blockchain explorer website designed specifically for the Ethereum network. It provides a set of APIs [13] that allow developers and researchers to interact with the Ethereum blockchain programmatically. However, `Etherscan` does not directly provide a list of all verified Solidity smart contracts on Ethereum. Hence, we needed to leverage `Dedaub` [12] first to retrieve a list of verified Solidity smart contract addresses from Ethereum. `Dedaub` is a web-based platform that provides a range of tools for analyzing and auditing smart contracts on the Ethereum network. It can search for newly deployed contracts on Ethereum and has the ability to search for verified Solidity smart contracts, including information on contract source code, byte code, disassembled code, contract ABI, etc. We developed a crawler for crawling verified Solidity smart contract addresses from the `Dedaub` Library website using its search engine starting from block number 16020730 and ending at the genesis block, which is the first block (a.k.a., block number one). On November 22nd, 2022, we successfully crawled 1057015 verified Solidity smart contract addresses with

different deployment transaction hashes from the Dedaub Library [12] which was available and traceable on the Etherscan website on that date. Then we developed another crawler using Etherscan APIs. This crawler searched for the content of these addresses and crawled the source code content from the Etherscan website. After that, we used MD5 calculation to remove contracts including identical source code and save only one copy for each duplicate contract set from this huge dataset. After removing duplicates, we ended up with a dataset of 35012 verified Solidity contracts. To ease the explanation, these 35012 contracts and 478 vulnerable cases together were called the Solidity contract dataset: $SC_{vulnerable}$ in the remaining part of the thesis.

4.2 Experimentation

The goal of this RQ was to compare the performance before and after tuning the tool configurations. Hence, we needed to obtain the clone detection results under the default setup as well as the outputs after the configuration tuning. For the tuning experiment, we followed the same process as in [98], which uses the Genetic Algorithm (GA) to locate the optimal setups. We first constructed a GA tuning dataset. Then we ran the GA algorithms on this dataset while varying the configurations of both tools. This process kept iterating until we obtained the optimal configurations which maximized the degree of overlap between the outputted clones of these two tools. For brevity, we called this approach a GAS-based approach in the remaining part of the thesis. .

GA Tuning Dataset We used the default configuration of NiCad and SourcererCC to check whether any of the 35012 verified contracts were clones with the vulnerable functions from $Code_{CVE}$. Among them, we randomly selected and manually verified a set of clone pairs to check if they are clones with the reported vulnerable functions. We repeated this process until we successfully verified 30 smart contracts, which were clones with existing vulnerable Solidity contracts from $Code_{CVE}$. Then we randomly selected another 470 Ethereum smart contracts from the overall 35012 smart contracts. We made sure that these 470 contracts were not the same as the previously selected 30 vulnerable contracts. In total, we selected 500 smart contracts as our GA tuning dataset. We intentionally produced such mixes as we wanted to ensure the resulting tool configurations can detect at least some clones from vulnerable functions so that we could use the same dataset for all the RQs.

We tuned the configurations of both tools by following the GAS-based approach described in [98]. NiCad has four tunable configuration parameters, namely: *minsize (linenumber)*, *maxsize (linenumber)*, *similarity threshold*, *rename*. The *minsize (linenumber)* here refers to the minimum number of lines in a structure in a pretty-printed line format using TXL grammar. The *maxsize (linenumber)* refers to the maximum number of lines with the same condition. The *similarity threshold* is the maximum different threshold the clone tools are set to be interested in. SourcererCC has three tunable configuration parameters: *min_ tokens*, *max_ tokens*, *similarity threshold* as indicated in Table 4.1. The *min_ tokens* refers to the minimum number of the token after parsing a code fragment to the token list that can be detected by SourcererCC. The *max_ tokens* refers to the maximum number of the token with

the same description as the minimum token. The *similarity threshold* refers to the similarity threshold set to filter clone pairs with a range of differences. The minimum length of the code fragment in SourcererCC is measured using a token number. Previous research showed that there is no simple correspondence between line numbers and token numbers in C and Java [98], in this thesis, we confirmed that there is also no simple correspondence between Solidity function line numbers and token numbers. We set the same range for the aforementioned tunable configuration parameters as in [98]. Then we encoded these tool configurations as the chromosome of our GA algorithm. Same as [98], we also set the individual count to be 100 for selecting the individual with the best agreement score, and the termination condition to be 100 generations with 100 populations inside one generation.

Same as [98], our fitness function f was defined to maximize the agreement score. The $AgreedFunc$ referred to the number of functions in an intersection reported by different clone detection tools. $ReportedFunc$ refers to the number of functions in a union reported by all clone tools on the same dataset. The Genetic Algorithm was guided by our fitness function in order to search the best set of parameters in a range of space Ω we defined. So given a configuration, our fitness function should return an agreement score indicating the degree of agreement made between different clone detection tools for an individual I. The formula for the fitness function is shown below:

$$f_I(TS(X), SC) = \frac{AgreedFunc}{ReportedFunc} \quad (X_I \in \Omega) \tag{4.1}$$

where X_I refers to the chromosome of an individual from the space Ω.

For setting up the initial population, we intentionally included the default configuration, the optimal configuration reported by [98] in the initial populations. For the remaining instances in our initial population, we used a random generation script to select configurations from Ω in order to form a wide-range population for doing mutation and crossover operations. In this way, we made sure that the default configuration and the previous optimal configurations from [98] were evaluated during this search and optimization process. For each instance in our population, we used a decoding script to parse the vector value into a set of configurations. If clone detection tools failed unexpectedly, we reset the tool to rerun the clone detection for that specific configuration. If any clone detection tool faced a non-terminate condition, we set a time limit of 10 minutes to do a rerun. We ran 100 generations for the GA algorithm to find the highest agreement score. Figure 4.1 showed the evolution of the agreement score after 100 generations. It seemed that the maximum agreement score and the average agreement score stabilized after 15 generations. This showed that we found the best set of configuration parameters in the end. We called the optimal configuration setup from this GA tuning process as the GAS-based configuration setup. Then we ran clone detection experiments on the GA tuning dataset under two different configuration setups: the default configuration setup and the GAS-based configuration setup. In Figure 4.1, we demonstrated the minimum, maximum, and average Agreement score for each generation.

4.2.1 Genetic Algorithm Results

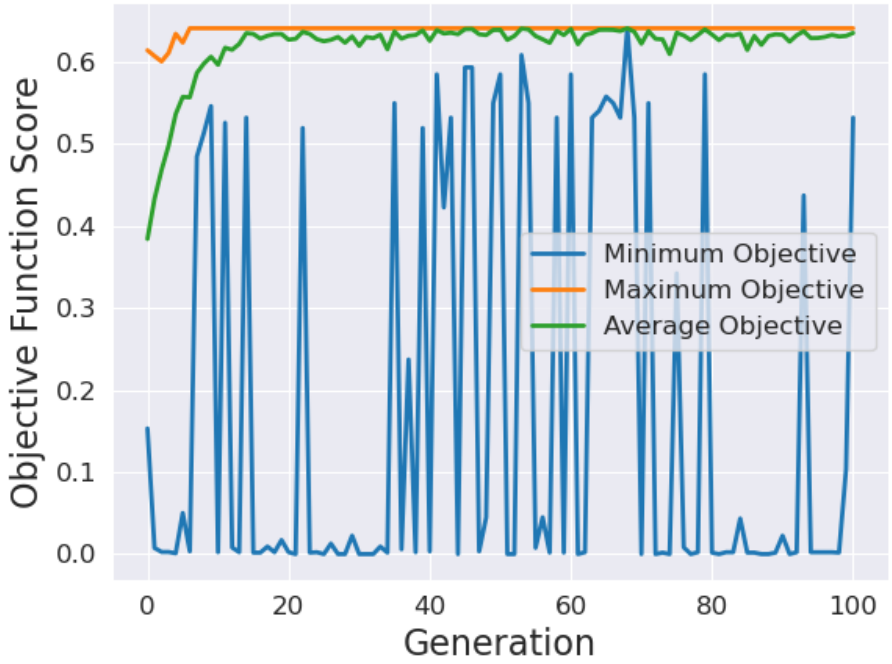


Figure 4.1: Agreement score in each generation for GA algorithm under GAS-based setup.

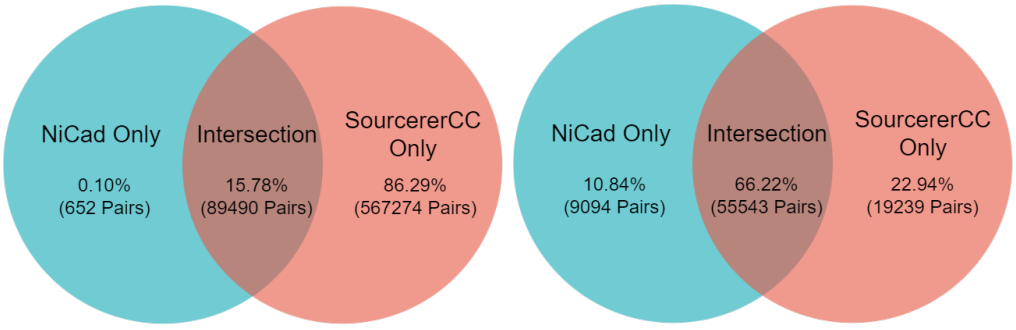
Tools	Parameter Name	Range	Configuration Settings	
			Default	General Agreement Score
NiCad	minsize (linenumber)	5-7	6	6
	maxsize (linenumber)	100-1000	1000	908
	similarity threshold	0.0-0.3	0.3	0
	rename	0-1	0	0
SourcererCC	min_tokens	10-300	16	28
	max_tokens	1000-3000	50000000	1021
	similarity threshold	7-10	8	10

Table 4.1: Optimal configuration outputted by the Genetic Algorithm under the GAS-based approach

4.3 Data Analysis

In this chapter, we conducted both quantitative and qualitative analysis for the clone detection experiments under the GAS-based and the default configuration setup from both tools.

4.3.1 Quantitative Analysis



(a) Clone Results using Default Configuration

(b) Clone Results using GAS-based Configuration

Figure 4.2: Venn Diagram for the Clone Pair Analysis for NiCad and SourcererCC.

We plotted two Venn diagrams for the union and intersection of clone pairs reported by both tools. As shown in Figure 4.2, we demonstrated (a) clone pairs reported by both tools using the default configuration and (b) clone pairs reported by both tools using the GAS-based configuration. As we can see, the degree of overlap and the amount of NiCad-only clones grew quite a bit after we tuned the configurations. However, there were still many clones reported only by SourcererCC and generally SourcererCC reported much more clones than NiCad.

4.3.2 Qualitative Analysis

We followed the grounded theory-based approach [25] to understand the rationale behind the different outputs from both tools under the default configuration setup. We first sampled the different code clones produced by either tool and manually analyzed them to understand the rationales. Then we developed scripts to automatically categorize such types and sampled another group of clones. We repeated this process until we successfully located all the reasons behind these differences. Then we repeated the same process for the clones produced under the GAS-based configuration setup. In the end, we successfully categorized the rationales behind all 567926 unique clone pairs for default configuration and 28333 unique clone pairs for GAS-based configuration reported in this RQ into the following six reasons:

[R1] Issues with TXL grammars on handling abstract functions. This type of clone was only reported by SourcererCC. The TXL grammar rules were inherited from Faizan et al. [55]. The raw Solidity code was parsed by these grammar rules. The results were saved into an intermediate XML file, which was used in the subsequent steps inside NiCad. However, the current TXL grammar rules removed abstract functions (a.k.a., functions without function bodies) from the potential clone list. Hence, they will not be processed in the subsequent steps and will not appear in the final clone detection results for NiCad. But the Ctags tool, which is used in the pre-processing step in SourcererCC, considers these abstract functions as

Reason	Clone Fragment 1	Clone Fragment 2
R1: Issues with NiCAD handling abstract functions using TXL grammar rules. Clone is in SourcererCC only	function swapTokensForExactTokens(uint amountOut, uint amountInMax, address[] calldata path, address to, uint deadline) external returns (uint[] memory amounts); **Contract Name:** SaninTv **Address:** 0x5e8785bb40a1d412fc69db473c9f731edd157cd0	function swapExactTokensForETH(uint amountIn, uint amountOutMin, address[] calldata path, address to, uint deadline) external returns (uint[] memory amounts); **Contract Name:** HAKURY **Address:** 0xefbE739c5B999C5Bca13be7Bb238610a3A8Fec93

R2: Issues with generating the correct begin/end line numbers from Ctags for Solidity functions. Clone is in NiCAD only	function decreaseApproval (address _spender, uint
	_subtractedValue) public returns (bool success) {
	uint oldValue = allowed[msg.sender][_spender];
	if (_subtractedValue > oldValue) {
	allowed[msg.sender][_spender] = 0;
	} else {
	allowed[msg.sender][_spender] =
	oldValue.sub(_subtractedValue);
	}
	emit Approval(msg.sender, _spender,
	allowed[msg.sender][_spender]);
	return true;
	}
	Contract Name: NLCToken
	Address: 0xdaf70f060dd0fe95d203bb620193d6283e6bf859
		function decreaseApproval (address _spender, uint
		_subtractedValue) public returns (bool) {
		uint oldValue = allowed[msg.sender][_spender];
		if (_subtractedValue > oldValue) {
		allowed[msg.sender][_spender] = 0;
		} else {
		allowed[msg.sender][_spender] =
		oldValue.sub(_subtractedValue);
		}
		emit Approval(msg.sender, _spender,
		allowed[msg.sender][_spender]);
		return true;
		}
		Contract Name: RBCToken
		Address: 0xcb7156fcc5eb8612cb3856b4f716e938ba761c94

R3: A small function with enough line numbers but not enough token numbers. Clone is in NiCad only	function withdraw() public onlyOwner nonReentrant{
	(bool os,) = payable (owner ()).call {
	value : address (this).balance
	}("");
	require (os);
	}
	Contract Name: GalacticExplorers
	Address: 0x3f7e4b1ebe37788a941dc095ed56c8ab2bb2d402
		function withdraw() public onlyOwner{
		(bool os,) = payable (owner ()).call {
		value : address (this).balance
		}("");
		require (os);
		}
		Contract Name: CINDYDAO
		Address:0x42D2e427Ed560650BA63ee1ea9186B270bf
		2Dc5b

R4: A small function with enough token numbers but not enough line numbers. Clone is in SourcererCC only	function functionCallWithValue(address target, bytes
	memory data, uint256 value) internal returns (bytes
	memory){
	return functionCallWithValue(target, data, value,
	"Address: low-level call with value failed");
	}
	Contract Name: EarnVesperStrategy
	Address: 0x3E281c4289b69573F0bD01Ab51C18A4f8144046f
		function functionCallWithValue(address target, bytes
		memory data, uint256 value) internal returns (bytes
		memory){
		return functionCallWithValue(target, data, value,
		"Address: low-level call with value failed");
		}
		Contract Name: LofiApeSociety
		Address: 0x4a520deea9204c7f7aef932b18548d187f6c68f2

R5: Threshold reach in the naive line-by-line text comparison similarity calculation from NiCAD but not in the overlap token similarity calculation from SCC. Clone is in NiCad only	function _transfer(address _from, address _to, uint _value)
	internal{
	require(_to != 0x0);
	require(balanceOf [_from] >= _value);
	require(balanceOf [_to] + _value > balanceOf [_to]);
	uint previousBalances = balanceOf[_from] + balanceOf[_to];
	balanceOf[_from] -= _value;
	balanceOf[_to] += _value;
	Transfer(_from, _to, _value);
	assert(balanceOf[_from] + balanceOf[_to] ==
	previousBalances);
	}
	Contract Name: IDRT
	Address: 0x34570cf88db31d4c518dee6057ff78e895dd80f1
		function _transfer(address _from, address _to, uint _value)
		internal{
		require(_to != 0x0);
		require(balanceOf[_from] >= _value);
		require(balanceOf[_to] + _value > balanceOf[_to]);
		require(!frozenAccount[_from]);
		require(!frozenAccount [_to]);
		balanceOf[_from] -= _value;
		balanceOf[_to] += _value;
		Transfer(_from, _to, _value);
		}
		Contract Name: IDRT
		Address: 0x34570cf88db31d4c518dee6057ff78e895dd80f1

R6: Threshold reach in the overlap token similarity calculation from SCC but not in the naive line-by-line text comparison token similarity calculation from NiCAD. Clone is in SourcererCC only	function mul(uint256 a, uint256 b) internal pure returns (uint256
	c){
	if (a == 0){
	return 0;
	}
	c = a * b;
	assert(c / a == b);
	return c;
	}
	Contract Name: PUBG
	Address: 0xc8aa6b089541824026c86091536673b5263b3f67
		function mul(uint256 a, uint256 b) internal pure returns
		(uint256){
		if (a == 0){
		return 0;
		}
		uint256 c = a * b;
		require(c / a == b);
		return c;
		}
		Contract Name: SchemeRegistrar
		Address: 0xc440d501b17347ccbc1a671c6164471898af38e7

Figure 4.3: Examples of unique clone pairs outputted from NiCad and SourcererCC.

functions and passes the processed results to the Solidity tokenizer. The first row (R1) from Figure 4.3 showed one such example.

[R2] Ctags issues on generating the correct begin/end line numbers for Solidity functions. This type of clone was only reported by NiCad, as we found that not all functions can be successfully generated by the Ctags grammar for SourcererCC. Ctags grammar, whose definition is obtained from [8], can identify most of the Solidity functions but fails to generate the correct begin/end line numbers for some Solidity functions. The second row (R2) from Figure 4.3 showed one such example.

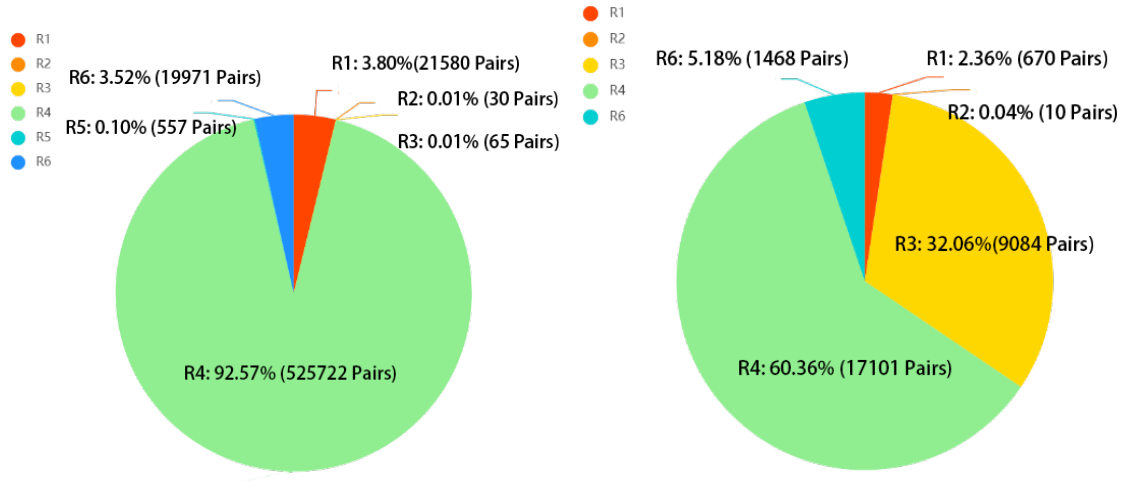
[R3] A small function with enough similar lines of code but not enough similar token numbers. This type of clone was only reported by NiCad. After pretty printing in NiCad preprocessing using TXL grammar, both of the functions in this clone pair has enough line numbers to do the comparison, and the similar line number reached the threshold of NiCad similarity. However, after SourcererCC's tokenization step, this clone pair does not reach the minimum token number threshold and hence won't appear in the subsequent similarity-matching process. In our experiment, we set the minimum token number to 28 tokens based on the GAS-based configuration. The default configuration for the minimum token number in SourcererCC is 16. This means that Solidity functions that with token numbers small than 16 (e.g., the example shown in the third row (R3) from Figure 4.3) were removed from the SourcererCC potential clone list.

[R4] A small function with enough similar token numbers but not enough similar lines of code. This type of clone was only reported by SourcererCC. After tokenizing this clone pair, both of the code snippets had enough token numbers to pass the minimum token number threshold of SourcererCC, and their similar token numbers reached the similarity threshold of SourcererCC. However, after NiCad pretty printing, this type of clone pair does not reach the minimum line number threshold of NiCad. The fourth row (R4) from Figure 4.3 showed one such example. In our experiment, we have the minimum line numbers to be 6 lines for both default and GAS-based configurations. This means that Solidity functions that with line numbers small than 6 (e.g., the example shown in the fourth row (R4) from Figure 4.3) were removed from the NiCad potential clone list.

[R5] Threshold reach in the naive line-by-line text comparison similarity calculation from NiCad but not in the overlap token similarity calculation from SourcererCC. This type of clone was only reported by NiCad. The fifth row (R5) from Figure 4.3 showed one such example. In this particular case, after pretty printing based on the TXL grammar, this pair of functions reached the similarity threshold of NiCad based on its similarity comparison algorithms (a.k.a., naive line-by-line text comparison). However, if they are calculated using the similarity comparison algorithm from SourcererCC (a.k.a., the overlap token similarity calculation), this pair of functions does not reach the similarity threshold for SourcererCC. Under the default configuration setup in this case,

the similarity threshold of SourcererCC was set to 80%. The similarity values of this sample code snippet were lower than the threshold and hence were not outputted by SourcererCC.

[R6] Threshold reach in the overlap token similarity calculation from SourcererCC but not in the naive line-by-line text comparison token similarity calculation from NiCad. This type of clone was only reported by SourcererCC. The sixth row (R6) from Figure 4.3 showed one such example. In this particular case, after the tokenizer step, this pair of functions reached the similarity threshold of SourcererCC using the overlap token similarity algorithm. However, if calculating based on the naive line-by-line text comparison similarity algorithm from NiCad, this pair of functions does not reach the similarity threshold for NiCad. Under the default configuration setup in this case, the similarity threshold for NiCad was set to 70%. The similarity values of this sample code snippet were lower than the threshold and hence will not be outputted by NiCad.



(a) Default configuration setup (b) GAS-based configuration setup

Figure 4.4: Pie chart showing reasons behind the unique clone pairs under default and GAS-based setup.

Figure 4.4 showed two pie charts with the breakdown of the rationales and their associated percentages of unique clone pairs under default and GAS-based setups in RQ1. The unique clones were attributed to six rationales under the default configuration setup. But under the GAS-based setup, the unique clones could be explained by the same rationales, except R5. In general, the number of unique clones was reduced under the GAS-based setup. This was due to the significant drop in clone pairs due to R4. As is shown in Figure 4.4a, under the default setup, the majority (92.57%) of the clone pairs were missed due to R4. The default threshold for the minimum token number of SourcererCC was 16. In order to maximize the overlap reported cloned functions, the GAS-based setup raised the minimum token number to 28 and set both of the similarity thresholds of clone detection tools to be 100%. As a

result, under the GAS-based setup, both tools only reported `Type-1` clones. This causes the clone pairs in R4 to drop sharply (96.75%).

Findings: The performance of these clone detection tools while analyzing Solidity contracts was quite different and can be quite different depending on their tool configurations. However, even after configuration tuning with the goal of maximizing the agreement between both tools, there were still 33.78% of the reported clone pairs, which could only be detected by just one of the tools. There are six main reasons behind such big differences in the cloning outputs.

Implications: Different tools detect different sets of clones and not all of the detected clones will lead to vulnerabilities. A thorough evaluation of these reported clones is needed in the context of vulnerability detection.

Chapter 5

RQ2 (Effectiveness Assessment)

In RQ1, we compared the performance of NiCAD and SourcererCC while optimizing for their general clone agreement score. However, not all of the detected clones led to vulnerabilities. Hence, in this RQ, we set to investigate the effectiveness of these two tools for detecting vulnerable Solidity contracts.

5.1 Data Collection

In the Data collection section, we used the configuration from RQ1 to produce our clone detection in RQ2. RQ1 was running on a small dataset with only 500 smart contracts. In RQ2, we mainly focused on detecting vulnerable clone pairs from $SC_{vulnerable}$. In other words, we want to detect clones between the downloaded 35012 smart contracts from RQ1 and $Code_{CVE}$ (a.k.a, 478 vulnerable case files) from the preliminary study (Chapter 3). Hence, no additional actions for data collection were needed in this RQ.

5.2 Experimentation

We ran clone detection using both tools with GAS-based configuration to detect clone pairs between the Ethereum smart contract code and the vulnerable cases. Clone detection tools may crash due to memory constraints when the dataset becomes large. Hence, we split 35012 smart contracts into 14 groups, each group containing 2500 contracts and 478 vulnerable case files. The last group contained 2512 contracts and 478 vulnerable case files.

5.3 Data Analysis

We have conducted quantitative analysis and qualitative analysis on the reported clones.

5.3.1 Quantitative Analysis

To evaluate the effectiveness of detecting software vulnerabilities, we calculated the precision, recall, and F1-scores on the resulting clones. Our quantitative analysis process was explained below. The results were shown in Table 5.1.

Precision: Some of the clone pairs might point to the same vulnerable functions. For example, the vulnerable functions reported CVE issues 1 and 2 are both clones of `functionA`. But since these two vulnerable functions are clones of each other and are reported in the same GitHub issue, both clone pairs essentially point to the same vulnerable issue. Hence, we only kept one such clone pair and removed all the others for the precision analysis. Then to cope with the large cloning output, we performed the statistical sampling approach to verify our results. We used the Sample Size Calculator [92] to determine the number of clone pairs that we needed to randomly extract from the whole vulnerable clone pair report. We randomly selected 382 and 265 cloned pairs from NiCad and SourcererCC, which correspond to 95% confidence level and ± 5 confidence interval.

For these samples, we manually went through every pair of the clones by deploying them on Remix [15] and tested them according to the specified steps reported in the vulnerable issues. Remix [15] is a website with an integrated development environment (IDE) specifically designed for writing, testing, and debugging Solidity smart contracts. We faithfully followed their steps from deploying contracts to testing and observing the results. If the same results were produced, this meant that the reported clones would suffer from the same vulnerability. In Figure 5.1, we demonstrated the case study samples for different types of clones. The left side is showing the reported vulnerable functions and the right side is showing the detected functions, which are clones of the left-side functions. For example, case 1 and case 2 are showing reported `Type-1` and `Type-2` clones with `CVE-2018-13218` [10] and `CVE-2018-13736` [11], respectively. Both reported cloned functions were verified to exhibit the same vulnerable behavior as the reported vulnerable issues. However, note that for case 3, which is a `Type-3` clone with `CVE-2018-10299` [9]. But after following the reported steps, the reported clone function turned out not to be a vulnerability. As shown in Table 5.1, the resulting precision for both tools under the GAS-based approach was quite good, with 0.9974 and 0.9895 for NiCad and SourcererCC, respectively.

Recall: Among all the 35012 Ethereum smart contracts, there were 435 contracts that were reported vulnerable contracts in the vulnerable issues. Although each of these vulnerable functions corresponds to one vulnerable report, some of these vulnerable issues were reported together. For example, GitHub issue [21] lists 354 vulnerable functions, which correspond to 354 different vulnerable issues. These functions were very similar or almost identical to each other. As developers already analyzed and tagged them as clones in this case, we grouped these GitHub issues into pairs of clones and used these pairs as our oracle for recall evaluation. We checked to see if these clone pairs were also reported as part of our clone detection results. As shown in Table 5.1, the resulting recall values for NiCad and SourcererCC were 0.2458 and 0.0002, respectively. We explained the rationale behind the low recall values in the qualitative analysis below.

F1-score: Based on the result we had from the precision and recall, we calculated the

Case Number	Reported Vulnerable Function	Source Code Function

Case 1:
Type 1 clone regarding CVE-2018-13218
Have verified CVE vulnerability
	function sell(uint256 amount) {
	if (balances[msg.sender] < amount) throw;
	balances[this] += amount;
	balances[msg.sender] -= amount;
	if (!msg.sender.send(amount * sellPrice)) {
	throw;
	} else {
	Transfer(msg.sender, this, amount);
	}
	}
	Contract Name: ICODollar
	Address: 0x7dfc425c0425d69ca577d76c6d53a26a6 c2cc0ac
		function sell(uint256 amount) {
		if (balances[msg.sender] < amount) throw;
		balances[this] += amount;
		balances[msg.sender] -= amount;
		if (!msg.sender.send(amount * sellPrice)) {
		throw;
		} else {
		Transfer(msg.sender, this, amount);
		}
		}
		Contract Name: MyAdvancedToken
		Address: 0x5aa4d0517d8f3be6060d2da5b 702e4 a462693ed7

Case 2:
Type 2 clone regarding CVE-2018-13736
Have verified CVE vulnerability
	function mintToken(address target, uint256 mintedAmount) onlyOwner public {
	balanceOf[target] += mintedAmount;
	totalSupply += mintedAmount;
	Transfer(0, this, mintedAmount);
	Transfer(this, target, mintedAmount);
	}
	Contract Name: ELearningCoinERC
	Address: 0x5d406656272da7c83966be544910732c4a547f86
		function mintToken(address target, uint256 initialSupply) onlyOwner public {
		balanceOf[target] += initialSupply;
		totalSupply += initialSupply;
		Transfer(0, this, initialSupply);
		Transfer(this, target, initialSupply);
		}
		Contract Name: AZTToken
		Address: 0x2e9f2a3c66ffd47163b362987765fd8857b1f3f9

Case 3:
Type 3 clone regarding CVE-2018-10299
Does not have verified CVE vulnerability
	function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns(bool) {
	uint cnt = _receivers.length;
	uint256 amount = uint256(cnt) * _value;
	require(cnt > 0 && cnt <= 20);
	require(_value > 0 && balances[msg.sender] >= amount);
	balances[msg.sender] = balances[msg.sender].sub(amount);
	for (uint i = 0; i < cnt; i++) {
	balances[_receivers[i]] = balances[_receivers[i]].add(_value);
	Transfer(msg.sender, _receivers[i], _value);
	}
	return true;
	}
	Contract Name: BecToken
	Address: 0xc5d105e63711398af9bbff092d4b6769c 82 f793d
		function batchTransfer(address[] _receivers, uint256 _value) public whenNotPaused returns(bool) {
		uint receiverCount = _receivers.length;
		uint256 amount = _value.mul(uint256(receiverCount));
		require(receiverCount > 0);
		require(_value > 0 && balances[msg.sender] >= amount);
		balances[msg.sender] = balances[msg.sender].sub(amount);
		for (uint i = 0; i < receiverCount; i++) {
		balances[_receivers[i]] = balances[_receivers[i]].add(_value);
		Transfer(msg.sender, _receivers[i], _value);
		}
		return true;
		}
		Contract Name: AtlantideToken
		Address: 0xe0ec671e941b5a1866b150148c75 78e2d68cd0ab

Figure 5.1: Clone pair samples drawn from the vulnerable clone pairs.

F1-score for both tools as shown in Table 5.1. The F1-score of SourcererCC is only 0.002 due to its low recall value.

Clone Detection Tools	Precision	Recall	F1-score
NiCad	0.9974	0.2458	0.3944
SourcererCC	0.9895	0.0001	0.0002

Table 5.1: Evaluation results for the GAS-based configuration setup based on the GA range defined in [98].

5.3.2 Qualitative Analysis

The qualitative analysis was mainly focused on analyzing the false positive cases during our precision analysis and reporting the rationales behind misreported issues during our recall analysis.

Clone type analysis: We classified the clone types in our manually analyzed 380 and 287 clone pair samples reported from NiCad and SourcererCC. We grouped these samples based on whether they were successfully identified as vulnerabilities or not. The results were shown in Figure 5.2. *Positive* means that the clone detection tool identifies as a vulnerable clone pair which is also verified to be correct (a.k.a., a real vulnerability). *Negative* means that although the tool identifies as a vulnerable clone pair, it was verified not to be an issue (a.k.a., false positive). As we can see from the table, we only had `Type-1` clones from the GAS-based setup. This was due to the similarity threshold from both tools being set to 100% in order to maximize the overlap function percentage. As shown in Figure 5.2, if there is a `Type-1` vulnerable clone pair identified, the chances to be a real vulnerability are quite high. 99.74% of all the detected `Type-1` clones for NiCad and 98.95% for SourcererCC turned out to be true vulnerabilities. Because all of the clones were `Type-1` clones, the precision for `Type-1` clones equals the precision of the vulnerability detection of this tool.

Misreported vulnerabilities: We manually went through the recall clonepairs to categorize the reasons why some of the vulnerable clones were not reported by either of the tools. In general, this part shared four of six reasons from RQ1. In our approach, we found that misreported recall pairs were due to 4 reasons (e.g., R3, R4, R5, and R6). To ensure consistency, we kept the same numbering (e.g., R3 and R4) while referring to these reasons. The results were visualized in a pie chart as shown in Figure 5.4. For SourcererCC, a total of 94050 clone pairs (57.01%) were missed. 94008 clone pairs (56.98%) were missed due to R3, which was the main reason behind most of the misreported vulnerable clone pairs between the two tools. 42 clone pairs (0.03%) were missed due to R5. For NiCad, there were 70930 (42.99%) clone pairs missed by NiCad. The reason for this was due to R4 and R6, which takes up 6036 pairs (3.66%) and 64894 pairs (39.33%) of all the misreported clone pairs.

To further investigate the low recall for SourcererCC, we calculated the source code characteristics for the vulnerable functions from the 478 vulnerable cases used in our study. It turns out that for these vulnerable functions, the minimum/median/maximum line number

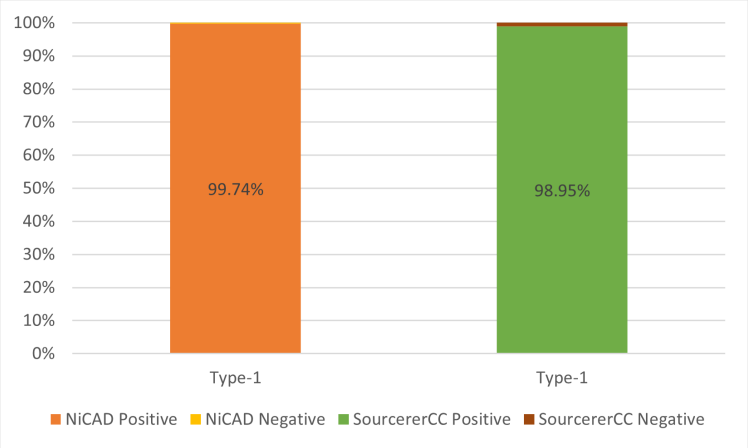


Figure 5.2: Stack Bar Chart Analysis

Figure 5.3: Clone type analysis under the GAS-based configuration setup

is 3/6/77. When considering the token numbers for these vulnerable functions, the minimum/median/maximum value is 6/21/246, respectively. The range of line number and the token number for these vulnerable Solidity functions was much lower than the range of the tunable configuration parameters set in the GA algorithms from the GAS-based approach shown in Table 4.1. The range that we chose for the GA tuning experiment was the same as [98], which was derived from conventional programming languages like Java and C. Note that the median of the minimum token number parameter value from the vulnerable functions is 21, which is much smaller than the value of this parameter (28) using the GAS-based tuning approach from RQ1. This was the main reason why most of the missing clone pairs were due to R3 from SourcererCC.

Findings: The effectiveness of these clone detection tools while detecting vulnerable Solidity contracts was quite different when we tuned the tool configurations based on the GAS-based approach. Under this approach, the effectiveness of SourcererCC was much worse than NiCad due to its super-low recall value. The main reason behind the sub-optimal performance from SourcererCC is due to its tool configuration issues.

Implications: Current GAS-based approach tunes the tool configurations simply based on the degree of agreement amongst different tools. The results from this RQ demonstrate that such an approach is too general, as they do not consider the appropriate usage context for vulnerabilities. In addition, the sub-optimal effectiveness is also due to the range of the configuration parameters set before the GA tuning process, as the code characteristics of the vulnerable Solidity functions are different from the conventional programming languages studied before. Hence, in the next RQ, we set to investigate the effectiveness of tuning the tool configurations while taking account into their specific usage contexts and code characteristics.

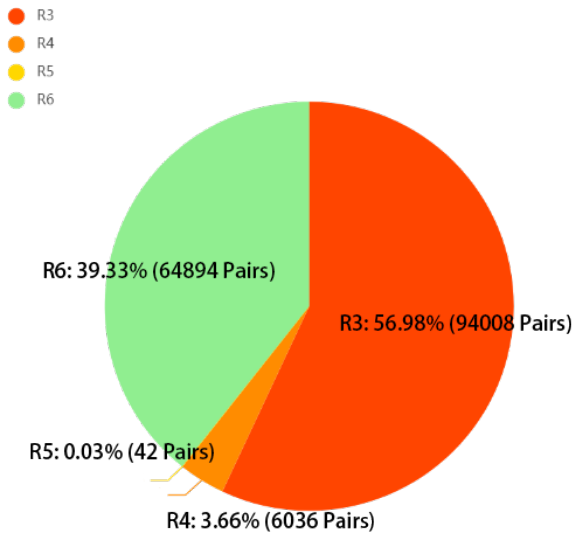


Figure 5.4: Pie chart showing reasons behind the misreported vulnerable clone pairs under the GAS-based setup.

Chapter 6

RQ3 (Context-specific Optimization)

The RQ2 results showed that, while detecting vulnerable Solidity contracts, the effectiveness of NiCAD and SourcererCC was not optimal if we only tuned their configurations based on the degree of agreement amongst their clone output based on the code characteristics of conventional programming languages. Hence, in this RQ, we designed new fitness functions for our GA approach while taking account into their context (a.k.a., usage context and code characteristics).

6.1 Data Collection

To ease comparison and ensure consistency of our results, we used the same GA tuning dataset and the same set of smart contracts (35012 smart contracts and 478 vulnerable case files, a.k.a, $SC_{vulnerable}$) from RQ2 in this RQ. No additional data collection actions were needed here.

6.2 Experimentation

In RQ2, we have demonstrated that some vulnerable cases could be lower than the threshold we used from Wang et al. [98]. This feature is derived from conventional programming languages like Java and C. So in RQ3, we lowered the minimum token number threshold and the minimum line number threshold to 6 and 3, which are the lowest threshold for reported vulnerable functions. This new configuration range was named Ω_L under the RQ3 GA setup. In this RQ, we reproduced our experiment on a GAS-based approach with the lower threshold Ω_L we defined. And we also designed the following three fitness functions which produce the context-specific GA tuning approach and take account into not only detecting more clones but also software vulnerabilities:

- *The Union objective function (O_u)*: In this objective function, we wanted to maximize the agreement score while detecting more vulnerable functions. We created a union (a.k.a., combining and removing the duplicates) of the detected vulnerable smart contract

clones from both tools. The size of this union dataset is called $UnionSC$. There are 30 contracts that have clones with vulnerable cases based on the data collection in RQ1 4. Hence, $\frac{UnionSC}{30}$ outputted a value between 0 and 1. Since the agreement score also outputted a value between 0 and 1, so this new objective function will always output a value between 0 and 1.

$$O_u = \frac{agreementscore * UnionSC}{30} \tag{6.1}$$

- *The Intersection objective function (O_i)*: To boost confidence and ensure result accuracy, developers or researchers generally preferred to cross-validate their results by comparing against more than one tool. Hence, in this objective function, we wanted to maximize the agreement score while detecting more common vulnerable functions between these two tools. We created an intersection (a.k.a., combining and only keeping the common ones) of the detected vulnerable smart contract clones from both tools. The size of this intersection dataset was called $IntersectionSC$. Since there were 30 vulnerable contracts in the GA tuning dataset. Hence, $\frac{IntersectionSC}{\#\ of\ known\ vulnerable\ functions}$ outputted a value between 0 and 1 and subsequently (O_i) always outputs a value between 0 and 1.

$$O_i = \frac{agreementscore * IntersectionSC}{30} \tag{6.2}$$

- *The Separate-tuning objective function (O_s)*: In this objective function, we wanted to tune each tool separately without focusing on the agreement score. The goal here was to see the maximum person each tool can achieve while conducting vulnerable clone detection. The number of vulnerable functions reported from each tool was denoted as $vulnerablecount$.

$$O_s = vulnerability\ count \tag{6.3}$$

In addition to the new objective functions, we have modified the search range of different configuration parameters compared to RQ2. As we have shown in RQ2, compared to traditional programming languages, the functions for some of these vulnerable Solidity contracts exhibited different characteristics (e.g., smaller lines of code or number of tokens). We ran these experiments and Table 6.1 shows the resulting tuned configuration parameter values under different approaches (a.k.a., GAS-based configurations and the aforementioned context-specific configurations). The resulting configuration values were different after tuning using different approaches. In the next section, we described our quantitative and qualitative analysis of the aforementioned four new configurations.

Tools	Parameter Name	Range	Configuration Settings			
			GAS	O_u	O_i	O_s
NiCad	minsize (linenumber)	3-7	3	3	3	4
	maxsize (linenumber)	100-1000	776	159	192	455
	similarity threshold	0.0-0.3	0	0.3	0.3	0.3
	rename	0-1	0	1	0	1
SourcererCC	min_tokens	6-100	8	11	8	11
	max_tokens	1000-3000	2668	1953	1665	2427
	similarity threshold	7-10	0	7	9	7

Table 6.1: The resulting optimal configurations after different GA tuning . GAS, O_u, O_i, O_s refer to the GA process while using the General-agreement-score/Union/Intersection/Separately tuned as the objective function, respectively.

6.3 Result Analysis

We conducted the same type of qualitative and quantitative analysis as in RQ2. The results were reported below.

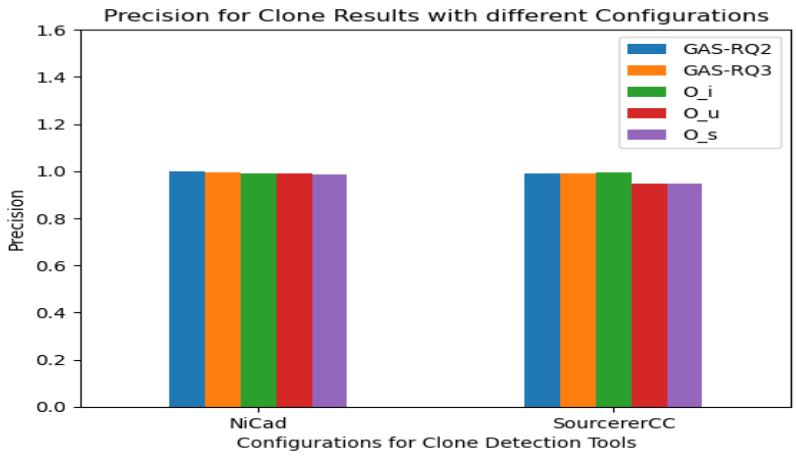
6.3.1 Quantitative Analysis

We followed the same process as the one described in RQ2 to calculate the precision/recall/F1-scores for different configuration setups under the aforementioned GA tuning processes. The results were shown in Figure 6.1(a), (b), and (c). For SourcererCC, the precision changed slightly (ranging from a 4.47% decrease to no change under GAS-based setups). For NiCad, the precision remained to be the same (ranging from a 1.05% decrease to no change under different setups). But note that the recall values are significantly higher when the objective functions were O_u and O_s for both tools. As a consequence, the F1-score values under O_u and O_s for SourcererCC were significantly better than the GAS-based approach (44 times). For NiCad, the best F1-score was achieved under the O_u-based approach, which was 1.64 times higher than the GAS-based approach. The best configuration setups for both SourcererCC and NiCad were achieved under the O_u-based approach. When comparing the two tools under their optimal setups, SourcererCC was much better than NiCad in recall (30.89%) and F1-scores (14.16%). NiCad's precision was a bit better than SourcererCC, but not much (4.70%). To conserve space, for the subsequent qualitative analysis part, we only focused on the optimal configuration setups for both tools.

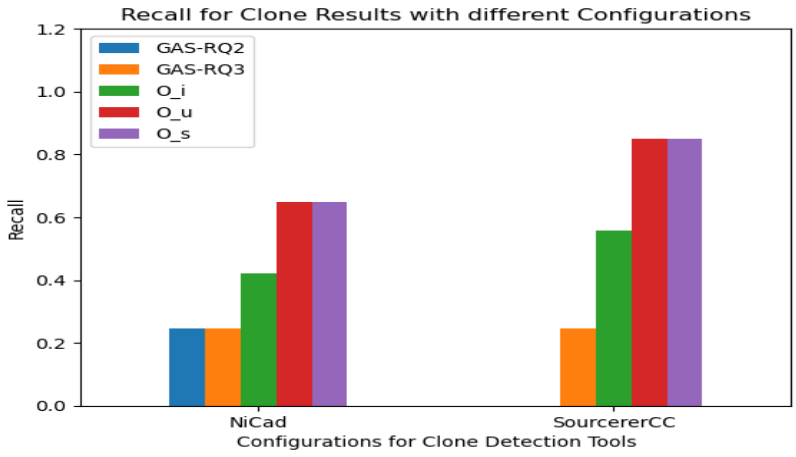
6.3.2 Qualitative Analysis

Same as RQ2, the qualitative analysis was focused on examining the false positive cases and the miss reported cases.

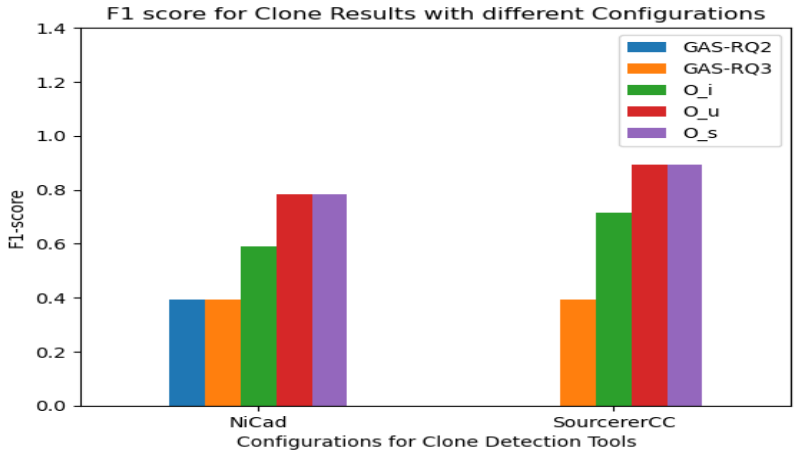
Clone type analysis: During the precision analysis, we have already manually verified



(a) Precision



(b) Recall



(c) F1-score

Figure 6.1: Precision, Recall, and F1-score under different configuration setups as a result of various GA tuning processes. ('default' stands for default configuration)

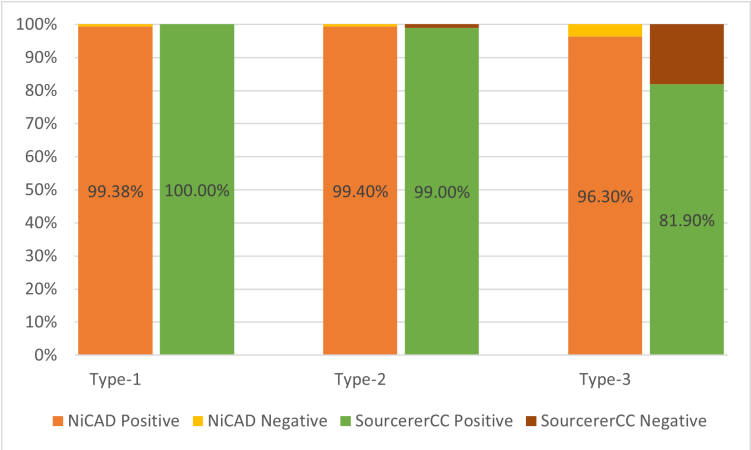


Figure 6.2: O_u-based result

Figure 6.3: Stack bar chart analysis for the positive and the negative cases of `Type-1, 2, 3` clones.

the randomly sampled 383 and another 383 clone pair samples from NiCad and SourcererCC already. Hence, we leveraged this dataset for our clone-type analysis. Same as RQ2, we classified the clone types for these random samples and grouped them based on whether they were successfully identified as vulnerabilities or not. The results were shown in Figure 6.3. As we can see from the figure, vulnerable functions could be either `Type-1, 2, or 3` clones. The falsely reported clones could also be `Type-1, 2` or `3` clones. While comparing the precision across different clone types, the precision for detecting `Type-1` and `2` clones for both tools under the O_u-based configuration setups were similar. However, for detecting vulnerable clones which are `Type-3` clones, we observed different results under these two configuration setups. Under O_u-based configuration setup, the precision of NiCad in detecting `Type-3` clones was higher than SourcererCC. As shown in Figure 6.4, the majority (86.28% to 81.90% for NiCad and SourcererCC) of all the verified vulnerabilities were `Type-2` and `3` clones for both tools. Under the GAS-based configuration setup in RQ2, all of the reported clones were `Type-1` clones under the default configuration setup. Under the optimal configuration setup (O_u), we enhanced the ability of both tools for detecting `Type-2` and `3` clones. In general, due to the additional `Type-2` and `3` clones detected, the recall for both tools improved quite a bit compared to the GAS-based setup in RQ2. Hence, subsequently, the F1-score also improves.

Recall investigation: We manually went through the recall clonepairs to categorize the reasons why some clones are not reported by these tools. Same as the previous RQs, we visualized the results in Figure 6.6. Compared to the GAS-based approach, the amount of misreported clone pairs has dropped (53.35% and 84.84% decrease in NiCad and SourcererCC, respectively) under the O_u-based configuration setups. The reason behind the significant improvement from SourcererCC was mainly due to the drop in the amount of misreported clones from R3. The two main reasons for the misreported clones for the O_u-based tuning approaches were: R5 and R6, which accounted for about 30% and 70% of the misreported

(a) O_u-based result for NiCad (b) O_u-based result for SourcererCC

Figure 6.4: Clone types of positive cases under the optimal configuration setup (O_u) for NiCad and SourcererCC.

clones respectively. We plotted a Venn diagram for all the reported clone pairs from both tools under O_u-based configuration setup. As we can see in Figure 6.5, most of the NiCad-detected clones were also detected by SourcererCC. However, in addition to the 74.31% common clones, there were also 24.51% unique clones detected by SourcererCC. Only 1.18% of the clones were uniquely identified by NiCad.

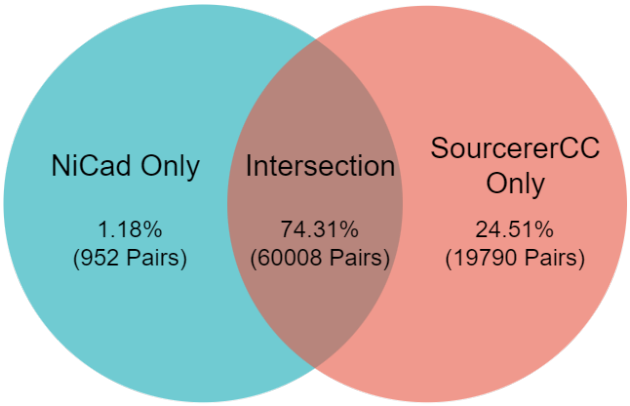


Figure 6.5: Venn diagram for reported recall pairs between NiCad and SourcererCC under O_u-based configuration setup.

In order to find the reason why NiCad was still worse than SourcererCC, we further looked into the rationale(s) behind why some of the clone pairs reported by SourcererCC were misreported by NiCad. Hence, it turns out the reason was due to the naive line-by-line text comparison similarity calculation from NiCad (a.k.a., R5). In other words, if the similarity was measured by the number of tokens (a.k.a., SourcererCC's approach), they reached the threshold. However, if the similarity was measured by lines of code (a.k.a., NiCad's approach), the similarity was below the threshold. Since there is also a confounding factor of the Solidity

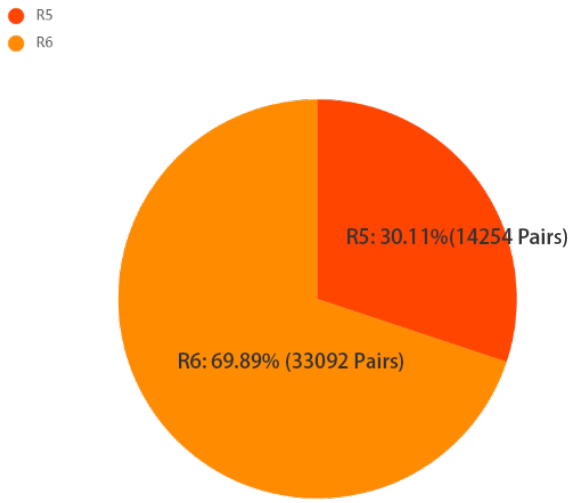


Figure 6.6: Misreported reason analysis for the O_u-based configuration setup.

grammar used in NiCad, we composed a similar segment of Java code, shown in Figure 6.7, and tried it on both NiCad and SourcererCC under the O_u setup. The same problem exists (a.k.a., clone pair only recognized by SourcererCC). Some possible mitigation approaches are considering both token and line similarity thresholds or possibly changing the similarity matching algorithms while accounting for the dense (more tokens per line) vs. sparse (fewer tokens per line) code.

Findings: The effectiveness of SourcererCC while detecting vulnerable Solidity contracts could be improved significantly (44 times better than the GAS-based approach in F1-score) if we tuned its configurations while considering its usage context and code characteristics. The impact of context-specific configuration tuning for NiCad was obvious (98.61% in F1-score compared to the GAS-based approach). When comparing the performance of both tools under their optimal configuration setups, SourcererCC was better than NiCad mainly due to the differences in similarity matching algorithms (token vs. lines).
Implications: Some clones could lead to vulnerabilities while others may not. Tuning the configurations of different clone detection tools while considering their context (usage context and code characteristics) may improve the effectiveness. Similarity can be measured in various different approaches which may work or not work in some specific usage context. Hence, future clone detection tools should improve their similarity measures so that they can operate in different contexts (e.g., dense vs. sparse code).

Language	Clone Fragment 1	Clone Fragment 2

Solidity Source Code	function sell (uint256 amount) public {
	if (sellPrice == 0) revert ();
	if (balanceOf [msg.sender] < amount) revert ();
	uint256 ethAmount = amount * sellPrice;
	balanceOf [msg.sender] -= amount;
	balanceOf [this] += amount;
	if (! msg.sender.send (ethAmount)) revert ();
	Transfer (msg.sender, this, amount);
	}
		function sell (uint256 amount) {
		if (balanceOf [msg.sender] < amount) throw;
		balanceOf [this] += amount;
		balanceOf [msg.sender] -= amount;
		if (! msg.sender.send (amount * sellPrice)) {
		throw;
		}
		else {
		Transfer (msg.sender, this, amount);
		}
		}

Converted Similar Java Code	public void sell(int amount) {
	if (sellPrice == 0)
	revert();
	if (balanceOf[msgsender] < amount)
	revert();
	int ethAmount = amount * sellPrice;
	balanceOf[msgsender] -= amount;
	balanceOf[this_number] += amount;
	if (!new MsgSender().send(ethAmount))
	revert();
	Transfer(new MsgSender(), this_number, amount);
	}
		void sell(int amount) {
		if (balanceOf[msgsender] < amount)
		throw_it();
		balanceOf[msgsender] -= amount;
		balanceOf[this_number] += amount;
		if (!new MsgSender().send(amount * sellPrice)){
		throw_it();
		}
		else {
		Transfer(new MsgSender(), this_number, amount);
		}
		}

Figure 6.7: Sample clone pairs written in Solidity and Java.

Chapter 7

Discussion and Future Work

In this chapter, we present some discussions based on the previous results from RQ1 to RQ3.

7.1 Other Clone Detection Tools

In this thesis, we focused on two widely used and representative clone detection tools: NiCad and SourcererCC. Although there are many clone detection techniques/tools proposed previously, there are very few suitable ones for our study after our assessment. We presented our assessment results below, which can highlight the gap and some future direction between research and practice in the area of clone-based vulnerability detection.

Source code-based: There are many other clone detection tools that can operate on the source code. However, based on our survey, only three works leverages existing open-source clone detection tools to study the code clones in the Solidity contract code: NiCad [55, 85] and Deckard [60]. Deckard [48] is also an AST-based clone detection tool. However, Deckard is out of the update and does not provide support for the new version of Solidity syntax that is later than its own update time, and they cannot recognize the method body structure under the new version of the syntax, which will result in missing clone pairs. Although Faizan et al. [55] showed that the code cloning results on Ethereum smart contracts from Deckard and NiCad were quite similar. But this research is conducted on the previous version of the Solidity contract code. We left tool updates and evaluations for these tools (e.g., Deckard or other text-based clone detection tools) as future work.

Bytecode-based: Some other clone detection tools can analyze the Solidity bytecode (e.g., Birthmark [66]). However, the published version of Birthmark raised errors and does not work for the new version of Solidity grammar which is later than its own update time. In addition to the lack of bytecode-based clone detection tools which can operate on the Solidity tool, we also found issues in terms of analyzing Solidity contract code at the bytecode level. This was mainly because most of the existing bytecode decompilation tool output does not closely resemble the original source code. For example, Figure 7.1b showed the actual source code public function and the decompiled source code public function by using two different decompilation tools: Online Solidity Decompiler [37] and the decompiled source code

from the Dedaub Library [12]. As shown in the figure, although the resulting decompiled public function and the structure do not closely resemble the original source code, their public method names (a.k.a., `balanceOf`) were the same and can be traced and found on Etherscan [36]. However, the method names for the internal functions are not the same between the original source code and the decompiled version. Figure 7.2 showed one such example. Although there are usage contexts that favor bytecode-level clone detection, due to the reasons stated here, we could not fully track the functions from bytecode to source code for verification and further analysis. Hence, we left the bytecode-level clone detection in the context of vulnerability detection as part of future research work.

```
214     function balanceOf(address account) public view override returns (uint256) {
215         return tokenFromReflection(_rOwned[account]);
216     }
```

(a) Source code function

```
function balanceOf(var arg0, var arg1) returns (var r0) {
    var var0 = 0x00;

    if (arg0 - arg1 i>= 0x20) {
        var var1 = msg.data[arg1:arg1 + 0x20];
        var var2 = 0x10c3;
        var var3 = var1;
        func_1AB0(var3);
        return var1;
    } else {
        var temp0 = var0;
        revert(memory[temp0:temp0 + temp0]);
    }
}
```

(b) Decompiled function from Online Solidity Decompiler

```
216   function balanceOf(address varg0) public nonPayable {
217       require(msg.data.length - 4 >= 32);
218       require(varg0 == varg0);
219       v0 = 0x593(varg0);
220       return v0;
221   }
```

(c) Decompiled function from the Dedaub Library

Figure 7.1: Public Function Example

7.2 Clone Management in the Context of Solidity Contract Development

Clone management refers to the process of identifying and handling code clones during the software development process. As we have shown in this thesis, all of the vulnerable clones were `Type-1`, `2`, and `3` clones. Such a problem was further exacerbated by the fact that a

```
function func_0376(var arg0) {
    if (msg.sender != storage[0x00] & (0x01 << 0xa0) - 0x01) {
        var temp0 = memory[0x40:0x60];
        memory[temp0:temp0 + 0x20] = 0x461bcd << 0xe5;
        var var1 = temp0 + 0x04;
        var var0 = 0x0442;
        var0 = manualswap(var1);
        var temp1 = memory[0x40:0x60];
        revert(memory[temp1:temp1 + var0 - temp1]);
    } else if (arg0 >= 0x19) { return; }
    else {
        storage[0x0c] = arg0;
        return;
    }
}
```

(a) Online Solidity Decompiler Example

```
350  function 0x593(uint256 varg0) private {
351      v0 = address(varg0);
352      require(owner_2[v0] <= stor_8, Error('Amount must be less than total reflections'));
353      v1, v2 = 0x13d6();
354      v3 = _SafeDiv(v1, v2);
355      v4 = _SafeDiv(v3, owner_2[v0]);
356      return v4;
357  }
```

(b) Dedaub Library Example

Figure 7.2: Internal Function Example

much high portion (about 79.2%) of the Ethereum smart contract code was clones compared to the conventional programming languages [60]. As a result of our analysis, we have also found 330 vulnerable functions which were clones of the existing reported vulnerable functions but were unreported. Due to the immutable nature of the deployed clone contracts, we recommend the following two actions: (1) for the existing deployed contracts, it is vital to conduct proactive clone management and detection to spot the impacted vulnerabilities sooner, and (2) while developing the new contract code, it is vital to adopt design patterns (e.g., the proxy pattern [35]) which enable flexibility and evolvability.

7.3 Vulnerability Discovery and Reporting

There are many other vulnerability detection approaches (e.g., fuzz testing, formal verification, symbolic execution, and static analysis) that can be used to detect vulnerabilities in the Ethereum smart contracts [104, 16]. One of the future works is to evaluate their effectiveness and integrate all these techniques while developing, testing, and deploying smart contracts. In addition, during our data curation processes, we found that there were many more vulnerable clones reported in the GitHub issues than in the vulnerable reports. This problem was also reported previously for other open-source projects for conventional programming languages (e.g., [78, 20]). Hence, advanced machine learning approaches (e.g., [41]) can be used complementary to our clone detection-based approaches.

7.4 Similarity Measurement

While developing clone detection tools, code similarities can be measured in different approaches (e.g., tokens, lines, or metrics). This can lead to unique clones detected only by some of the techniques but not others. In our work, we have found that the main reasons behind misreported vulnerable clone pairs under optional configuration setups are mainly due to R5 and R6, which were caused by discrepancies in terms of similarity measurements using tokens (a.k.a., SourcererCC's approach) or lines (a.k.a., NiCad's approach). Previous work [98] reported that they did not find a strong connection between token numbers and line numbers in programming languages like Java and C. This was also true for Solidity based on our experiments in this thesis, as some lines may contain many tokens while other lines may only have one or two tokens. Depending on the usage context (e.g., testing, code refactoring, or vulnerability detection), different clone pairs are needed. It would be very difficult for practitioners to pick some clone detection tools which suit their needs the best. Hence, one of the future directions for code cloning research would be researching and incorporating context-specific similarity measures in different clone detection techniques, so that their clone detection results converge.

Chapter 8

Threats to Validity

This chapter discusses the threats related to this paper.

8.1 External Validity

We have selected two clone detection tools, namely NiCad and SourcererCC. NiCad is using AST-based clone detection techniques, whereas SourcererCC is a token-based technique. Although both techniques are representative, our results cannot be generalized to other clone detection techniques, such as other text-based, tree-based, and graph-based clone detection techniques in the context of Ethereum smart contract vulnerabilities. In addition, as we have discussed in the Discussion chapter 7 that bytecode-level clone detection techniques are not mature enough. Hence, we focused on source code level clone detection. Our results cannot be generalized to clone detection on other types of software artifacts. Finally, this was a case study focusing on detecting vulnerable Ethereum smart contract clones. Since the characteristics of the Solidity contract code are quite different from conventional programming languages like C and Java [60, 75], the findings reported in this thesis cannot be generalized to clone-based vulnerability detection in other programming languages.

8.2 Internal Validity

As shown in [98] and [100], the effectiveness of clone detection techniques depends not only on the approaches but also on the configurations of these tools. Tuning the configuration could improve the performance of the clone detection tool to detect more clone pairs. In this work, we have experimented with a set of tool configuration tuning approaches on two different clone detection approaches while keeping all the other factors constant.

8.3 Construct Validity

We followed the same process as many other existing works (e.g., [81] [18]) to calculate the precision/recall/F1-score values while evaluating the performance of clone detection in the context of vulnerability detection.

Chapter 9

Conclusions

In this paper, we have studied the effectiveness of detecting software vulnerabilities while using clone detection techniques. We applied this in the context of Ethereum smart contracts, as previous studies [60, 55] showed a much higher portion of the cloned code deployed in the Ethereum platform compared to conventional software projects. The effectiveness of these clone detection tools improved significantly (up to 44 times improvement in F-1 score compared to the GAS-based configurations) if we tuned their tool configurations based on their usage context. As part of this process, we have discovered 330 vulnerable smart contracts which are previously unknown. Although there were common clones reported from different clone detection tools, some unique clones can only be reported by one of the studied tools due to their internal differences in terms of similarity measurement algorithms. This thesis highlighted the need for further research in the area of context-specific clone detection and management research in the future. In future work, we plan to further improve the clone detection techniques by researching into context-specific similarity measurements. We also plan to look into automated code repair techniques to fix the detected vulnerable clone pairs.

Bibliography

[1] Solidity ast parser. https://github.com/ConsenSys/python-solidity-parser, November 2021.

[2] CVE-2018-12959. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-12959, 2022. Accessed on 18 Jan 2023.

[3] CVE-2018-14085. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-14085, 2022. Accessed on 18 Jan 2023.

[4] CVE-2018-18425. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-18425, 2022. Accessed on 18 Jan 2023.

[5] CVE-2018-20178. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-20178, 2022. Accessed on 18 Jan 2023.

[6] CVE-2022-24788. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-24788, 2022. Accessed on 18 Jan 2023.

[7] CVE-2022-35936. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-35936, 2022. Accessed on 18 Jan 2023.

[8] Ctag rules for solidity. https://gist.github.com/shuangjj/ae816cacffce3a27e256de7c21312c50, 2022.

[9] CVE-2018-10299. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10299, 2022. Accessed on 18 Jan 2023.

[10] CVE-2018-13218. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-13218, 2022. Accessed on 18 Jan 2023.

[11] CVE-2018-13736. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-13736, 2022. Accessed on 18 Jan 2023.

[12] Dedaub library. https://library.dedaub.com/, 2022.

[13] Etherscan api website. https://docs.etherscan.io/, 2022.

[14] John Aach, Prashant Mali, and George M Church. Casfinder: Flexible algorithm for identifying specific cas9 targets in genomes. *BioRxiv*, page 005074, 2014.

[15] Rana M. Amir Latif, Khalid Hussain, N. Z. Jhanjhi, Anand Nayyar, and Osama Rizwan. Retracted article: A remix ide: smart contract-based framework for the healthcare sector by using blockchain technology. *Multimedia Tools and Applications*, 81(19): 26609–26632, Aug 2022. ISSN 1573-7721. doi: 10.1007/s11042-020-10087-1. URL `https://doi.org/10.1007/s11042-020-10087-1`.

[16] Imran Ashraf and W. K. Chant. An empirical study on the effects of entry function pairs in fuzzing smart contracts. In *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pages 1716–1721, 2022. doi: 10.1109/COMPSAC54236.2022.00273.

[17] Hamid Abdul Basit, Damith C. Rajapakse, and Stan Jarzabek. Beyond templates: A study of clones in the stl and some general implications. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, page 451–459, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139632. doi: 10.1145/1062455.1062537. URL `https://doi.org/10.1145/1062455.1062537`.

[18] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007. doi: 10.1109/TSE.2007.70725.

[19] Nicolas Bettenburg, Weyi Shang, Walid Ibrahim, Bram Adams, Ying Zou, and Ahmed E. Hassan. An empirical study on inconsistent changes to code clones at release level. In *2009 16th Working Conference on Reverse Engineering*, pages 85–94, 2009. doi: 10.1109/WCRE.2009.51.

[20] Asaf Biton. Responsible disclosure: the impact of vulnerability disclosure on open source security. `https://snyk.io/blog/responsible-disclosure/`.

[21] BlockChainsSecurity and Gang Li. Ethertokens. `https://github.com/BlockChainsSecurity/EtherTokens`, 2018.

[22] Tamara Brandstätter, Stefan Schulte, Jürgen Cito, and Michael Borkowski. Characterizing efficiency optimizations in solidity smart contracts. In *2020 IEEE International Conference on Blockchain (Blockchain)*, pages 281–290, 2020. doi: 10.1109/Blockchain50366.2020.00042.

[23] Rainer Böhme, Nicolas Christin, Benjamin Edelman, and Tyler Moore. Bitcoin: Economics, technology, and governance. *Journal of Economic Perspectives*, 29(2):213–38, May 2015. doi: 10.1257/jep.29.2.213. URL `https://www.aeaweb.org/articles?id=10.1257/jep.29.2.213`.

[24] Pedro M. Caldeira, Kazunori Sakamoto, Hironori Washizaki, Yoshiaki Fukazawa, and Takahisa Shimada. Improving syntactical clone detection methods through the use of an intermediate representation. In *2020 IEEE 14th International Workshop on Software Clones (IWSC)*, pages 8–14, 2020. doi: 10.1109/IWSC50091.2020.9047637.

[25] Kathy Charmaz. *Constructing Grounded Theory: A Practical Guide Through Qualitative Analysis*, volume 1. 01 2006.

[26] Huashan Chen, Marcus Pendleton, Laurent Njilla, and Shouhuai Xu. A survey on ethereum systems security: Vulnerabilities, attacks, and defenses. *ACM Comput. Surv.*, 53(3), jun 2020. ISSN 0360-0300. doi: 10.1145/3391195. URL `https://doi.org/10.1145/3391195`.

[27] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. *SIGOPS Oper. Syst. Rev.*, 35(5):73–88, oct 2001. ISSN 0163-5980. doi: 10.1145/502059.502042. URL `https://doi.org/10.1145/502059.502042`.

[28] James R. Cordy. The txl source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2006.04.002. URL `https://www.sciencedirect.com/science/article/pii/S0167642306000669`. Special Issue on The Fourth Workshop on Language Descriptions, Tools, and Applications (LDTA '04).

[29] J.R. Cordy. Comprehending reality - practical barriers to industrial adoption of software maintenance automation. pages 196– 205, 06 2003. ISBN 0-7695-1883-4. doi: 10.1109/WPC.2003.1199203.

[30] J.R. Cordy, C.D. Halpern, and E. Promislow. Txl: a rapid prototyping system for programming language dialects. In *Proceedings. 1988 International Conference on Computer Languages*, pages 280–285, 1988. doi: 10.1109/ICCL.1988.13075.

[31] Chris Dannen. *Introducing Ethereum and Solidity*. 01 2017. ISBN 978-1-4842-2534-9. doi: 10.1007/978-1-4842-2535-6.

[32] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. An empirical investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 65:127–139, 02 2003. doi: 10.1016/S0164-1212(02)00054-7.

[33] Ignatios Deligiannis, Ioannis Stamelos, Lefteris Angelis, Manos Roumeliotis, and Martin Shepperd. A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software*, 72(2):129–143, 2004. ISSN 0164-1212. doi: https://doi.org/10.1016/S0164-1212(03)00240-1. URL `https://www.sciencedirect.com/science/article/pii/S0164121203002401`.

[34] Andrea Di Sorbo, Sonia Laudanna, Anna Vacca, Corrado A. Visaggio, and Gerardo Canfora. Profiling gas consumption in solidity smart contracts. *J. Syst. Softw.*, 186(C), apr 2022. ISSN 0164-1212. doi: 10.1016/j.jss.2021.111193. URL `https://doi.org/10.1016/j.jss.2021.111193`.

[35] Amir M Ebrahimi, Bram Adams, Gustavo A Oliva, and Ahmed E Hassan. A large-scale exploratory study on the proxy pattern in ethereum.

[36] Etherscan. Etherscan—the ethereum blockchain explorer. *https://etherscan. io/*, 2021.

[37] ethervm@gmail.com. Online solidity decompiler. `https://ethervm.io/decompile`.

[38] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 313–324, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450330138. doi: 10.1145/2642937.2642982. URL `https://doi.org/10.1145/2642937.2642982`.

[39] Farima Farmahinifarahani, Vaibhav Saini, Di Yang, Hitesh Sajnani, and Cristina V Lopes. On precision of code clone detection tools. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 84–94, 2019. doi: 10.1109/SANER.2019.8668015.

[40] Michael Fröwis and Rainer Böhme. In code we trust? In Joaquin Garcia-Alfaro, Guillermo Navarro-Arribas, Hannes Hartenstein, and Jordi Herrera-Joancomartí, editors, *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 357–372, Cham, 2017. Springer International Publishing. ISBN 978-3-319-67816-0.

[41] Michael Gegick, Pete Rotella, and Tao Xie. Identifying security bug reports via text mining: An industrial case study. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 11–20, 2010. doi: 10.1109/MSR.2010.5463340.

[42] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. Binary code clone detection across architectures and compiling configurations. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, page 88–98. IEEE Press, 2017. ISBN 9781538605356. doi: 10.1109/ICPC.2017.22. URL `https://doi.org/10.1109/ICPC.2017.22`.

[43] Sungjae Hwang and Sukyoung Ryu. Gap between theory and practice: An empirical study of security patches in solidity. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 542–553, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371216. doi: 10.1145/3377811.3380424. URL `https://doi.org/10.1145/3377811.3380424`.

[44] Judith Islam, Manishankar Mondal, Chanchal Roy, and Kevin Schneider. A comparative study of software bugs in clone and non-clone code. pages 436–443, 07 2017. doi: 10.18293/SEKE2017-056.

[45] Judith F. Islam, Manishankar Mondal, and Chanchal K. Roy. A comparative study of software bugs in micro-clones and regular code clones. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 73–83, 2019. doi: 10.1109/SANER.2019.8667993.

[46] Md Rakibul Islam, Minhaz F. Zibran, and Aayush Nagpal. Security vulnerabilities in categories of clones and non-cloned code: An empirical study. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 20–29, 2017. doi: 10.1109/ESEM.2017.9.

[47] Reza Jelveh. Universal ctags. `https://github.com/universal-ctags/ctags`, 2022. Accessed on 2 June 2022.

[48] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, page 96–105, USA, 2007. IEEE Computer Society. ISBN 0769528287. doi: 10.1109/ICSE.2007.30. URL `https://doi.org/10.1109/ICSE.2007.30`.

[49] Zigui Jiang, Xiuwen Tang, Zibin Zheng, Jinyan Guo, Xiapu Luo, and Yin Li. Calling relationship investigation and application on ethereum blockchain system. *Empirical Software Engineering*, 28(2):31, Jan 2023. ISSN 1573-7616. doi: 10.1007/s10664-022-10240-4. URL `https://doi.org/10.1007/s10664-022-10240-4`.

[50] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495, 2009. doi: 10.1109/ICSE.2009.5070547.

[51] Huzefa Kagdi, Malcom Gethers, Denys Poshyvanyk, and Michael L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *2010 17th Working Conference on Reverse Engineering*, pages 119–128, 2010. doi: 10.1109/WCRE.2010.21.

[52] C. Kapser, M. Godfrey, R. Al-Ekram, and R. Holt. Cloning by accident: an empirical study of source code cloning across software systems. In *2005 International Symposium on Empirical Software Engineering*, page 10 pp., Los Alamitos, CA, USA, nov 2005. IEEE Computer Society. doi: 10.1109/ISESE.2005.1541846. URL `https://doi.ieeecomputersociety.org/10.1109/ISESE.2005.1541846`.

[53] Cory J. Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13(6):

645–692, Dec 2008. ISSN 1573-7616. doi: 10.1007/s10664-008-9076-6. URL https://doi.org/10.1007/s10664-008-9076-6.

[54] Saruhan Karademir, Thomas Dean, and Sylvain Leblanc. Using clone detection to find malware in acrobat files. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '13, page 70–80, USA, 2013. IBM Corp.

[55] Faizan Khan, Istvan David, Daniel Varro, and Shane McIntosh. Code cloning in smart contracts on the ethereum platform: An extended replication study. *IEEE Transactions on Software Engineering*, 49(4):2006–2019, 2023. doi: 10.1109/TSE.2022.3207428.

[56] Miryung Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Proceedings. 2004 International Symposium on Empirical Software Engineering, 2004. ISESE '04.*, pages 83–92, 2004. doi: 10.1109/ISESE.2004.1334896.

[57] Miryung Kim, Vibha Sazawal, David Notkin, and Gail Murphy. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*, 30(5):187–196, sep 2005. ISSN 0163-5948. doi: 10.1145/1095430.1081737. URL https://doi.org/10.1145/1095430.1081737.

[58] Seulbae Kim and Heejo Lee. Software systems at risk: An empirical study of cloned vulnerabilities in practice. *Comput. Secur.*, 77(C):720–736, aug 2018. ISSN 0167-4048. doi: 10.1016/j.cose.2018.02.007. URL https://doi.org/10.1016/j.cose.2018.02.007.

[59] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, 2017. doi: 10.1109/SP.2017.62.

[60] Masanari Kondo, Gustavo A. Oliva, Zhen Ming (Jack) Jiang, Ahmed E. Hassan, and Osamu Mizuno. Code cloning in smart contracts: a case study on verified contracts from the ethereum blockchain platform. *Empirical Software Engineering*, 25(6):4617–4675, Nov 2020. ISSN 1573-7616. doi: 10.1007/s10664-020-09852-5. URL https://doi.org/10.1007/s10664-020-09852-5.

[61] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*.

[62] Wei Li and Raed Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *J. Syst. Softw.*, 80(7):1120–1128, jul 2007. ISSN 0164-1212. doi: 10.1016/j.jss.2006.10.018. URL https://doi.org/10.1016/j.jss.2006.10.018.

[63] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3): 176–192, 2006. doi: 10.1109/TSE.2006.28.

[64] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: An automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC '16, page 201–213, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450347716. doi: 10.1145/2991079.2991102. URL `https://doi.org/10.1145/2991079.2991102`.

[65] Zhou Liao, Shuwei Song, Hang Zhu, Xiapu Luo, Zheyuan He, Renkai Jiang, Ting Chen, Jiachi Chen, Tao Zhang, and Xiaosong Zhang. Large-scale empirical study of inline assembly on 7.6 million ethereum smart contracts. *IEEE Transactions on Software Engineering*, 49(2):777–801, 2023. doi: 10.1109/TSE.2022.3163614.

[66] Han Liu, Zhiqiang Yang, Yu Jiang, Wenqi Zhao, and Jiaguang Sun. Enabling clone detection for ethereum via smart contract birthmarks. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, page 105–115. IEEE Press, 2019. doi: 10.1109/ICPC.2019.00024. URL `https://doi.org/10.1109/ICPC.2019.00024`.

[67] Jing Liu and Zhentian Liu. A survey on security verification of blockchain smart contracts. *IEEE Access*, 7:77894–77904, 2019. doi: 10.1109/ACCESS.2019.2921624.

[68] Yinghang Ma. Solidity-function-level-tokenizer-for-sourcerercc, June 2023. URL `https://github.com/Coppelian/Solidity-function-level-tokenizer-for-SourcererCC`.

[69] Yinghang Ma and Faizan Khan. solidity-nicad, 2022. URL `https://github.com/eff-kay/solidity-nicad/tree/sol_test`.

[70] Naouel Moha and Yann-Gael Guéhéneuc. Decor: A tool for the detection of design defects. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 527–528, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595938824. doi: 10.1145/1321631.1321727. URL `https://doi.org/10.1145/1321631.1321727`.

[71] Debajani Mohanty. Ethereum for architects and developers: With case studies and code samples in solidity. *Ethereum for Architects and Developers: With Case Studies and Code Samples in Solidity*, pages 105–138, 01 2018.

[72] Manishankar Mondal, Chanchal Roy, and Kevin Schneider. Bug-proneness and late propagation tendency of code clones: A comparative study on different clone types. *Journal of Systems and Software*, 144, 05 2018. doi: 10.1016/j.jss.2018.05.028.

[73] M.J. Munro. Product metrics for automatic identification of "bad smell" design problems in java source-code. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 9 pp.–9, 2005. doi: 10.1109/METRICS.2005.3.

[74] Kentaro Ohno, Norihiro Yoshida, Wenqing Zhu, and Hiroaki Takada. On the effectiveness of clone detection for detecting iot-related vulnerable clones. *arXiv preprint arXiv:2110.10493*. URL https://doi.org/10.48550/arXiv.2110.10493.

[75] Gustavo A. Oliva, Ahmed E. Hassan, and Zhen Ming (Jack) Jiang. An exploratory study of smart contracts in the ethereum blockchain platform. *Empirical Software Engineering*, 25(3):1864–1904, May 2020. ISSN 1573-7616. doi: 10.1007/s10664-019-09796-5. URL https://doi.org/10.1007/s10664-019-09796-5.

[76] Michael Pacheco, Gustavo Oliva, Gopi Krishnan Rajbahadur, and Ahmed Hassan. Is my transaction done yet? an empirical study of transaction processing times in the ethereum blockchain platform. *ACM Trans. Softw. Eng. Methodol.*, 32(3), apr 2023. ISSN 1049-331X. doi: 10.1145/3549542. URL https://doi.org/10.1145/3549542.

[77] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. Detecting bad smells in source code using change history information. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE '13, page 268–278. IEEE Press, 2013. ISBN 9781479902156. doi: 10.1109/ASE.2013.6693086. URL https://doi.org/10.1109/ASE.2013.6693086.

[78] Shengyi Pan, Jiayuan Zhou, Filipe Roseiro Cogo, Xin Xia, Lingfeng Bao, Xing Hu, Shanping Li, and Ahmed E. Hassan. Automated unearthing of dangerous issue reports. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 834–846, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549156. URL https://doi.org/10.1145/3540250.3549156.

[79] Germán Poo-Caamaño and Bradley M. Kuhn. ccfinderx. https://github.com/gpoo/ccfinderx, 2018.

[80] Chanchal K. Roy and James R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *2008 16th IEEE International Conference on Program Comprehension*, pages 172–181, 2008. doi: 10.1109/ICPC.2008.41.

[81] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009. ISSN 0167-6423. doi: https://doi.org/10.1016/j.scico.2009.02.007. URL https://www.sciencedirect.com/science/article/pii/S0167642309000367.

[82] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[83] Dilan Sahin, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. Code-smell detection as a bilevel problem. *ACM Trans. Softw. Eng. Methodol.*, 24(1), oct 2014. ISSN 1049-331X. doi: 10.1145/2675067. URL `https://doi.org/10.1145/2675067`.

[84] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, page 1157–1168, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339001. doi: 10.1145/2884781.2884877. URL `https://doi.org/10.1145/2884781.2884877`.

[85] Noama Fatima Samreen and Manar H. Alalfi. Volcano: Detecting vulnerabilities of ethereum smart contracts using code clone analysis, 2022.

[86] Gehan M.K. Selim, King Chun Foo, and Ying Zou. Enhancing source-based clone detection using intermediate representation. In *2010 17th Working Conference on Reverse Engineering*, pages 227–236, 2010. doi: 10.1109/WCRE.2010.33.

[87] Majd Soud, Grischa Liebel, and Mohammad Hamdaqa. A fly in the ointment: An empirical study on the characteristics of ethereum smart contracts code weaknesses and vulnerabilities, 2022.

[88] Jeffrey Svajlenko and Chanchal K. Roy. Evaluating modern clone detection tools. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 321–330, 2014. doi: 10.1109/ICSME.2014.54.

[89] Jeffrey Svajlenko and Chanchal K. Roy. Evaluating clone detection tools with bigclonebench. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–140, 2015. doi: 10.1109/ICSM.2015.7332459.

[90] Jeffrey Svajlenko and Chanchal K Roy. A survey on the evaluation of clone detection performance and benchmarking. *arXiv preprint arXiv:2006.15682*, 2020. URL `https://doi.org/10.48550/arXiv.2006.15682`.

[91] Melanie Swan. *Blockchain: Blueprint for a New Economy*. O'Reilly Media, Inc., 1st edition, 2015. ISBN 1491920491.

[92] Creative Research Systems. Sample size calculator. `https://www.surveysystem.com/sscalc.htm`, 1982. Accessed on 10 June 2022.

[93] Nikolaos Tsantalis, Davood Mazinanian, and Giri Panamoottil Krishnan. Assessing the refactorability of software clones. *IEEE Transactions on Software Engineering*, 41(11): 1055–1090, 2015. doi: 10.1109/TSE.2015.2448531.

[94] Anna Vacca, Michele Fredella, Andrea Di Sorbo, Corrado A. Visaggio, and Gerardo Canfora. An empirical investigation on the trade-off between smart contract readability and gas consumption. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*, pages 214–224, 2022. doi: 10.1145/3524610.3529157.

[95] Brent van Bladel and Serge Demeyer. A comparative study of test code clones and production code clones. *Journal of Systems and Software*, 176:110940, 2021. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2021.110940. URL `https://www.sciencedirect.com/science/article/pii/S0164121221000376`.

[96] Brent van Bladel and Serge Demeyer. A comparative study of code clone genealogies in test code and production code. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 913–920, 2023. doi: 10.1109/SANER56733.2023.00110.

[97] Common Vulnerabilities. Common vulnerabilities and exposures. `https://cve.mitre.org/`, 2005. Accessed on 10 June 2022.

[98] Tiantian Wang, Mark Harman, Yue Jia, and Jens Krinke. Searching for better configurations: A rigorous approach to clone evaluation. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 455–465, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322379. doi: 10.1145/2491411.2491420. URL `https://doi.org/10.1145/2491411.2491420`.

[99] Rui Xi. *A large-scale empirical study of low-level function use in Ethereum smart contracts and automated replacement*. PhD thesis, University of British Columbia, 2022. URL `https://open.library.ubc.ca/collections/ubctheses/24/items/1.0416021`.

[100] Yong Shi Boyuan Chen Zhen Ming (Jack) Jiang Yan Zhou, Jinfu Chen. An empirical comparison on the results of different clone detection setups for c-based projects. 2023.

[101] Xiao Yi, Daoyuan Wu, Lingxiao Jiang, Yuzhou Fang, Kehuan Zhang, and Wei Zhang. An empirical study of blockchain system vulnerabilities: Modules, types, and patterns. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 709–721, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450394130. doi: 10.1145/3540250.3549105. URL `https://doi.org/10.1145/3540250.3549105`.

[102] Ma Yinghang. Solidity_Exploratory, June 2023. URL `https://github.com/Coppelian/Solidity_Exploratory`.

[103] Abdullah A. Zarir, Gustavo A. Oliva, Zhen M. (Jack) Jiang, and Ahmed E. Hassan. Developing cost-effective blockchain-powered applications: A case study of the gas usage of smart contract transactions in the ethereum blockchain platform. *ACM Trans.*

Softw. Eng. Methodol., 30(3), mar 2021. ISSN 1049-331X. doi: 10.1145/3431726. URL https://doi.org/10.1145/3431726.

[104] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Lin. Demystifying exploitable bugs in smart contracts. ICSE, 2023.

[105] Minhaz F Zibran and Chanchal K Roy. The road to software clone management: A survey. *Dept. Comput. Sci., Univ. of Saskatchewan, Saskatoon, SK, Tech. Rep*, 3, 2012.