CloneCompass: Visualizations for Code Clone Analysis

by

Ying Wang
B.Sc., University of Electronic Science and Technology of China, 2013
M.Sc., University College London, 2014

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

CloneCompass: Visualizations for Code Clone Analysis

by

Ying Wang
B.Sc., University of Electronic Science and Technology of China, 2013
M.Sc., University College London, 2014

Supervisory Committee

Dr. Margaret-Anne Storey, Supervisor
(Department of Computer Science)

Dr. Daniel M. German, Departmental Member
(Department of Computer Science)

ABSTRACT

Code clones are identical or similar code fragments in a single software system or across multiple systems. Frequent copy-paste-modify activities and reuse of existing systems result in maintenance difficulties and security issues. Addressing these problems requires analysts to undertake code clone analysis, which is an intensive process to discover problematic clones in existing software. To improve the efficiency of this process, tools for code clone detection and analysis, such as Kam1n0 and CCFinder, were created.

Kam1n0 is an efficient code clone search engine that facilitates assembly code analysis. However, Kam1n0 search results can contain millions of function-clone pairs, and efficiently exploring and comprehensively understanding the resulting data can be challenging. This thesis presents a design study whereby we collaborated with analyst stakeholders to identify requirements for a tool that visualizes and scales to millions of function-clone pairs. These requirements led to the design of an interactive visual tool, CloneCompass, consisting of novel TreeMap Matrix and Adjacency Matrix visualizations to aid in the exploration of assembly code clones extracted from Kam1n0.

We conducted a preliminary evaluation with the analyst stakeholders, and we show how CloneCompass enables these users to visually and interactively explore assembly code clones detected by Kam1n0 with suspected vulnerabilities. To further validate our tool and extend its usability to source code clones, we carried out a Linux case study, where we explored the clones in the Linux kernel detected by CCFinder and gained a number of insights about the cloning activities that may have occurred in the development of the Linux kernel.

Contents

Supervisory Committee **ii**

Abstract **iii**

Table of Contents **iv**

List of Tables **vii**

List of Figures **viii**

Acknowledgements **x**

Dedication **xi**

1 Introduction **1**

1.1 Outline 3

2 Background **6**

2.1 Reasons for Code Clones 6

2.2 Consequences of Code Clones 7

2.3 Clone Types and Relations 8

2.4 Kam1n0 9

2.4.1 Kam1n0 Search Process 9

2.4.2 Kam1n0 Clone-Search Applications 11

3 Related Work **13**

3.1 Visualization for Code Clone Analysis 13

3.1.1 Node-link diagrams 14

3.1.2 Matrix-based views 16

3.1.3 A Hybrid Approach 18

3.2 Matrix Views for Large-Scale Data 18
3.2.1 Zoomable Adjacency Matrix 19

4 Research Methodology **22**
4.1 Design Study 23
4.1.1 Knowledge Base 23
4.1.2 Problem Characterization 25
4.1.3 Development of Artifacts 25
4.1.4 Evaluation 26

5 Problem Characterization **28**
5.1 Existing visualizations used in Kam1n0 28
5.2 Visualization Challenges in Kam1n0 30

6 Solution Development: CloneCompass **32**
6.1 Visualization Design Process 32
6.2 TreeMap Matrix – An Overview 34
6.2.1 Examples of TreeMap Matrices 34
6.2.2 Tooltips in a TreeMap 36
6.2.3 The use of pagination 37
6.2.4 Multiple selections 38
6.3 Adjacency Matrix – A Detailed View 39
6.3.1 Zooming and Panning 40
6.3.2 Glyphs and Color Schemes 40
6.3.3 Filters 42
6.4 Workflow 43
6.5 Implementation 44

7 Preliminary Evaluation of CloneCompass **45**
7.1 Datasets and Participants 45
7.2 Procedure 46
7.3 Findings 47
7.3.1 Support for Many-to-Many Comparisons 47
7.3.2 Support for Scalability of Clone Exploration 47
7.3.3 To Provide a Big Picture Understanding of How Clones Are Dispersed Across Many Systems 48

7.3.4 User Experience 49
7.3.5 Refining User Requirements 50

8 Linux Case Study **52**
8.1 Data Preparation 52
8.2 Linux Case Study Findings 53
8.2.1 Aggregated by File Size 53
8.2.2 Aggregated by Number of Contributors 56
8.2.3 Aggregated by Number of Commits 58
8.2.4 Aggregated by Year Created 59

9 Discussion and Future Work **60**
9.1 Discussion of Preliminary Evaluation and Linux Case Study Results . 60
9.2 Limitations and Future Work 61

10 Conclusions **64**

A Binning Process **66**

B Adjacency Matrix Using Different Color Schemes **68**

C Questions **70**

D Kam1n0 Search Result Example **71**

Bibliography **72**

List of Tables

Table 2.1 Types of code clones 8

List of Figures

Figure 1.1 Overview - TreeMap Matrix 4
Figure 1.2 Detail - Adjacency Matrix: it shows the data filtered by the rectangles in TreeMap Matrix in Fig. 1.1 5

Figure 2.1 The process of a clone search in Kam1n0 10

Figure 3.1 Tree-based diagrams for code clone analysis 15
Figure 3.2 Network diagrams for code clone analysis 16
Figure 3.3 Adjacency Matrix [1] 17
Figure 3.4 Matrix-based views used in D-CCFinder [2] 18
Figure 3.5 Zooamble Adjacency Matrix 19

Figure 4.1 Design study research methodology applied in this thesis: the arrows above the blocks and overlaps between two stages imply the iterative dynamics of the stages 24
Figure 4.2 Concept of Histogram Matrix 26

Figure 5.1 Detailed view in Kam1n0: (a) target functions list; (b) a detailed view shows a target function and its clones; (c) a node-edge graph view shows a target function and its clones 29
Figure 5.2 Summary view in Kam1n0 30

Figure 6.1 Design process of visualizations 33
Figure 6.2 TreeMap Matrix ordered and aggregated by code size 35
Figure 6.3 TreeMap Matrix aggregated by version 36
Figure 6.4 Hover and tooltip in TreeMap Matrix 37
Figure 6.5 TreeMap Matrix with pagination 38
Figure 6.6 Multiple selections of inner and outer rectangles in TreeMap Matrix 39
Figure 6.7 Adjacency Matrix before and after zooming/panning 40

Figure 6.8 "Hovered Function-Clone Pair" section in an Adjacency Matrix 41
Figure 6.9 FatFont in Adjacency Matrix 41
Figure 6.10 Color scheme choices and filters in Adjacency Matrix 42
Figure 6.11 Filters in Adjacency Matrix 42

Figure 8.1 TreeMap Matrix aggregated by file size 55
Figure 8.2 Adjacency Matrix ordered by the number of tokens in Linux files 56
Figure 8.3 TreeMap Matrix aggregated by the number of contributors . . 57
Figure 8.4 TreeMap Matrix aggregated by number of commits 58
Figure 8.5 TreeMap Matrix aggregated by year created 59

Figure B.1 Adjacency Matrix using BrBG color scheme 68
Figure B.2 Adjacency Matrix using BrBG color scheme 69
Figure B.3 Adjacency Matrix using Viridis color scheme 69

Figure D.1 An example of a search result generated by Kam1n0 71

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deep and sincere gratitude to my supervisor, **Dr. Margaret-Anne (Peggy) Storey**, for her mentorship, enthusiasm, and encouragement throughout all stages of my study and research. Without her invaluable guidance and inspiration, this thesis would not have been possible.

I would also like to thank **Dr. Daniel M. German** for his support, insightful feedback, and extended discussion, which have contributed to the improvement of this thesis.

To all **present and former members of the CHISEL lab**, I am grateful for your suggestions on this thesis, our friendship, and all the fun we have had: Alexey Zagalsky, Andreas Koenzen, Arman Yousefzade, Carly Lebeuf, Courtney Bornholdt, Eirini Kalliamvakou, Huihui (Nora) Huang, Jorin Weatherston, Leon Li, Matthieu Foucault, Omar Elazhary, Trishala Bhasin, Neil Ernst, and Cassandra Petrachenko. A special thanks to **Matthieu** for sharing his insights, knowledge, and experience that greatly assisted my research; **Jorin** for providing helpful advice and generous support for my work; **Cassie** for her immediate help when I needed it and her thoughtful editing of this thesis as well as other documents.

I was fortunate to collaborate with **Martin Salois, Steven H.H. Ding, and Benjamin C.M. Fung** in my research. I appreciate their great support, valuable feedback, and participation in the evaluation of this research.

Lastly, I would like to thank my **parents, husband, and friends** for their love, understanding, and support along the way.

DEDICATION

To my parents and my husband.

Chapter 1

Introduction

Developers commonly copy and paste a piece of code from one place to another with or without modifications. A survey [3] was conducted by Zhang et al. who asked developers how often they apply such copy-paste-modification practices in their work. The survey results show that all participants copied code: 33% of those surveyed reported that they often do; 67% stated that they sometimes do. Such frequent copy-paste-modification activities are reasons for code clones in software systems. Another reason for code cloning is because of the evolutionary nature of software development. For example, a large portion of code or a software system is copied and used as a "springboard" to ensure new versions evolve independently from old versions [4], which is also called forking [5].

Since code cloning brings many benefits, such as productivity increases, this activity is unavoidable and uncontrollable at times [6]. Frequent cloning leads to many problems such as increased maintenance difficulties, software plagiarism and copyright infringement issues, and vulnerability propagation. For example, if a piece of code contains an unknown fault, the fault could be propagated to any clones of the piece of code, which may result in security problems. Therefore, many studies focus on code clone detection and analysis to solve problems caused by code cloning. Although researchers could manually detect and analyze code clones, it is often challenging and time-consuming, so many automatic tools have been created to reduce this manual work. More details on code cloning and existing clone detection tools are discussed in Chapter 2.

Although source code detection and analysis are commonly discussed in the literature, when source code is unavailable, the granularity of code needs to be scaled

down to the assembly level. Kam1n0[1] is an existing assembly code clone search engine mainly used to detect security vulnerabilities caused by code cloning. In the security analysis of binaries, a security analyst aims to identify security vulnerabilities in the assembly being inspected. Typically, analysts have to inspect many different applications, each of which can consist of millions of assembly level instructions. Using Kam1n0, analysts can efficiently find similar assembly functions from multiple binaries, and identify potential vulnerabilities that have been cloned.

Kam1n0 provides some initial visualizations to support clone inspections. This tool presents its results in three different ways:

1. **Function subgraph views:** A flow graph view, a text diff view, and a clone group view are used to compare the subgraphs of two functions. These views are useful for comparing the details of two functions when analysts search for clones of a given function.

2. **Detailed view:** A detailed view shows the similarities between a target function (i.e., a function given as a query) and its clones.

3. **Summary view:** A summary view shows statistics about the clones of two binaries.

Kam1n0's features are quite effective at helping analysts inspect clones and assess their potential vulnerabilities. However, a large software ecosystem can contain a very large number of clone pairs. For example, the search results of several chromium-based software systems can contain over 90 million clone pairs. Using the current detailed and summary views in Kam1n0 is challenging when inspecting a large number of clones. What is needed is a high-level view of the cloning of a set of software binaries, and the ability to navigate and inspect clones in specific regions of the clone dataset.

In this thesis I describe a visual tool for code clone analysis called CloneCompass. CloneCompass implements a pairing of two visualizations for very large clone sets: a novel visualization TreeMap Matrix (see Fig. 1.1) and an Adjacency Matrix (see Fig. 1.2). CloneCompass provides many-to-many comparisons for large-scale datasets and enables reverse engineers[2] to efficiently explore clones identified by Kam1n0. Throughout an exploration, reverse engineers can gain insights about the software ecosystem under analysis and more efficiently find vulnerabilities, which is critical in

[1] https://github.com/McGill-DMaS/Kam1n0-Community

[2] I use the terms reverse engineers and security analysts interchangeably throughout the thesis.

security analysis. To identify Kam1n0 users' needs and to find an efficient visual solution to address their issues, I conducted a design study [7] with Kam1n0's creators, who are researchers in the Data Mining and Security (DMaS) Lab at McGill University,[3] and Kam1n0's end users, who are reverse engineers from Defence Research and Development Canada (DRDC).

The main contributions of this thesis are as follows:

- **Design of CloneCompass:** This study proposed a design of a novel TreeMap Matrix view, which coordinates with an Adjacency Matrix for clone analysis. The TreeMap Matrix and the combination of a TreeMap Matrix and an Adjacency Matrix may also be used for the code analysis and the duplicated information analysis by future researchers.

- **Problem characterization:** The problems of code clone analysis from our stakeholders were identified and abstracted in the Problem Characterization stage of my design study. Such problems are only used in this study but could also be used by future researchers who may suggest different solutions to the same problem.

- **Preliminary evaluation:** A preliminary evaluation was conducted to validate the usability of CloneCompass in assembly code clone analysis. The evaluation demonstrated that CloneCompass can help users obtain useful insights, and reflected the future improvements of this study.

- **Linux case study:** A Linux case study was conducted to extend the applicability of CloneCompass. A number of insights about Linux were identified in the case study, which demonstrated that CloneCompass is potentially capable of analyzing clones in other domains.

1.1 Outline

This thesis is organized as follows:

Chapter 2 gives an introduction of code clones, including reasons for code cloning and the corresponding consequences, and the types and relations of clones. This

[3] http://dmas.lab.mcgill.ca/

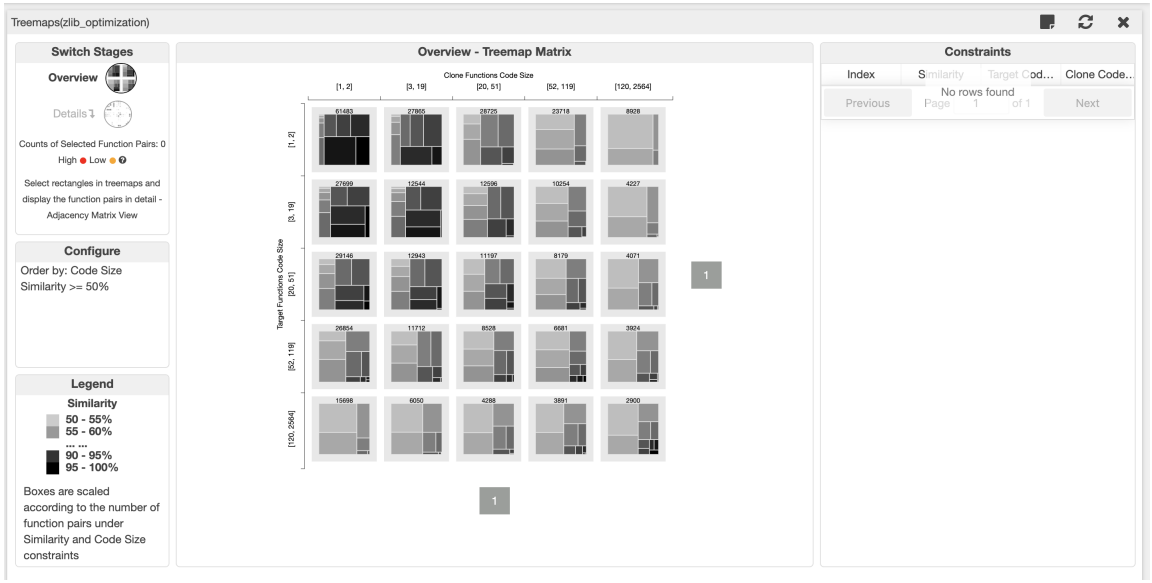


Figure 1.1: Overview - TreeMap Matrix

chapter also gives an introduction of Kam1n0, including its search process and the *clone-search applications* it provides.

Chapter 3 describes the state-of-the-art visualization techniques for code clone analysis, including node-link diagrams, matrix-based views, and hybrids of the two views. Existing matrix-based views for large-scale data are also discussed in order to increase the scalability of typical matrix-based views.

Chapter 4 contains my research methodology—a design study. This chapter introduces the design study and illustrates my tasks at each stage of the study.

Chapter 5 describes the existing visualizations used in Kam1n0, and characterizes the problems that our collaborators have when they use Kam1n0's visualizations to explore their clone datasets.

Chapter 6 outlines my proposed approach to solve our collaborators' problems (identified in Chapter 5). This chapter describes the visualization design process and the features of the TreeMap Matrix and the Adjacency Matrix in CloneCompass. The workflow of CloneCompass and how I implemented it are also covered in this chapter.

Chapter 7 discusses the preliminary evaluation I conducted to validate CloneCompass, including datasets, participants, procedures, and the findings of this eval-

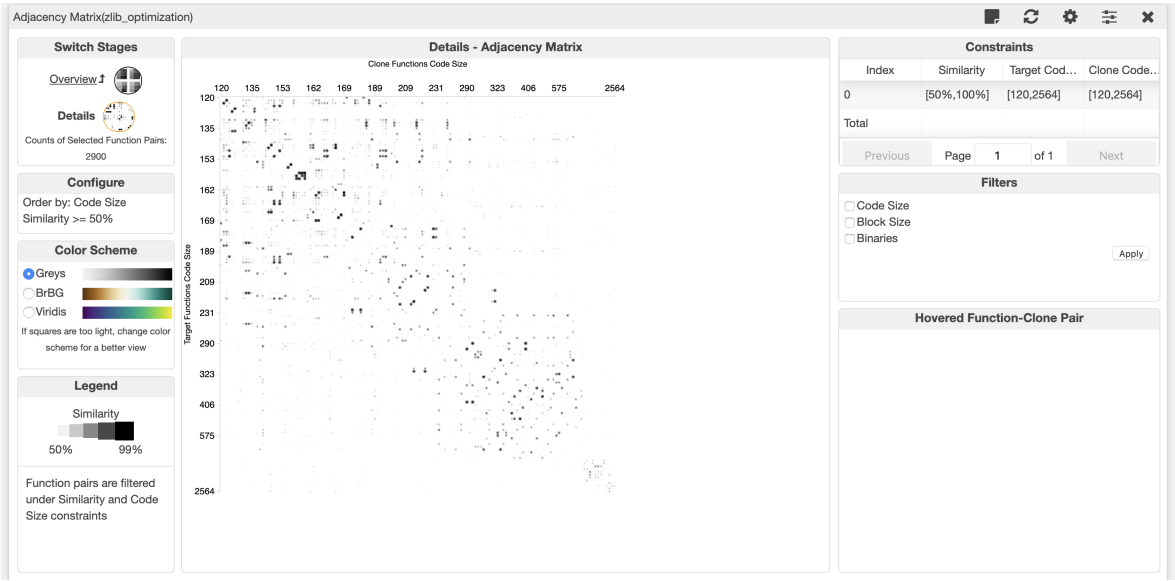


Figure 1.2: Detail - Adjacency Matrix: it shows the data filtered by the rectangles in TreeMap Matrix in Fig. 1.1

uation.

Chapter 8 describes how CloneCompass can be applied to visualizing the clones in the Linux kernel.

Chapter 9 discusses the strengths and limitations of my study, and points to possible future work.

Chapter 10 summarizes the results of my study and draws conclusions.

Chapter 2

Background

Code clones are identical or similar code fragments introduced by the reuse of code in software programs. While code cloning has its benefits—reusing existing code can increase productivity or reduce mistakes when implementing similar functionality—it can also cause problems such as maintenance difficulties [6] and the propagation of vulnerabilities. In this chapter, I first give the reasons for code clones, and explain the benefits and the problems that code cloning brings to software engineering. Next, I briefly describe clone types and relations. Finally, I give an introduction to Kam1n0, as our stakeholders have chosen Kam1n0 to search for clones in multiple binaries.

2.1 Reasons for Code Clones

Code clones can be introduced in several ways. They are often created by copy-paste-modify activities in programming practice, i.e., copying existing code and pasting it somewhere else with or without modification [6][3]. Additionally, the reuse of designs, functionalities, and logic also give rise to code clones [8][9]; for example, developers may copy and then modify a working subsystem to port a driver in Linux [10]. However, clones can also accidentally appear. The reasons for intentional and accidental code clones are summarized below.

Intentional cloning

Developers copy existing code in order to gain benefits. Copying and pasting can save time and effort when the existing code has already been used to solve current problems because developers do not need to write new code or refactor existing code [4].

Besides, developing new code and refactoring can bring risks [11], and code cloning can ensure good performance in real-time programs [3].

Another benefit of code cloning relates to the performance of developers. For example, sometimes code clones are created because lines of code (LOC) are used to evaluate a developer's productivity [12].

A primary benefit of code cloning concerns maintenance needs. For example, an entire system can be copied to guarantee new versions evolve independently from previous versions [4]. Additionally, code clones can maintain robustness in software architectures. For instance, similar design patterns can increase the readability and quality of a system [4].

The limitations of developers' knowledge is another reason for code clones. When developers have limited programming design skills, they often search for solutions to similar problems [13]. When developers have limited understanding of large systems, which are generally difficult to understand, developers could use existing code as an example. Such code clones probably are introduced by learning and experimenting at the beginning, but then developers may forget or do not have time to remove the clones they created. Also, when developers are not aware of the harmfulness of code clones, they may think a quick copy does not cause any problems [3].

Accidental cloning

Accidental clones may be created when developers do not know a piece of code already exists in the system. Also, developers tend to use similar solutions to solve similar problems they have addressed in the past, so clones may be found in code written by the same person. In addition, using the same libraries and APIs can cause code clones [8][14].

2.2 Consequences of Code Clones

Although code cloning provides many benefits, this activity also brings a number of drawbacks:

1. Clones that contain bad design elements (e.g., a system with a lack of abstractions) may increase the difficulty maintaining software systems [8]. As a result, developers have to spend more time understanding and maintaining such systems.

2. A piece of code with bugs can be propagated throughout multiple software systems by code cloning practices [6][4], possibly causing significant security issues.

3. Duplicated work is required when a piece of code needs to be updated, because developers need to keep track of all the clones and update them accordingly. For instance, if faults are found in a piece of code that has been cloned, all its clones need to be inspected and corrected, which increases maintenance costs and requires more time and effort from developers [8][15].

In conclusion, code cloning is a common practice in software engineering, bringing both advantages and disadvantages. In the next section, I discuss the different types of clones.

2.3 Clone Types and Relations

Generally, researchers classify code clones based on textual and functional resemblance [12][16][17]. Textual clones are code fragments that are similar in syntax or expression, including Type I, II, and III. If code fragments are similar in functionality or semantics (e.g., functions with different syntax performing the same logic), the clones are classified as Type IV. Table 2.1 describes four types of code clones. Type I represents clones without modifications except for variations in blank spaces, comments, and layouts. Type II clones are syntactically similar with only changes in identifiers, such as names of variables and classes. Type III includes clones with further modifications, such as statements which are added, deleted, or changed. Type IV represents clones that result from semantic similarities of code fragments. Kam1n0 supports the detection of all types of clones, which are discussed in the next section.

Type I	Clones without modifications except for variations in blank spaces, comments, and layouts.
Type II	Clones that are syntactically similar with only changes in identifiers, such as names of variables, classes.
Type III	Clones with further modifications, such as statements that are added, deleted, or changed.
Type IV	Clones that result from semantic similarity of code fragments.

Table 2.1: Types of code clones

In code clone literature, researchers often define clone relations based on different levels of abstraction [9].

- **A Clone Pair** is a pair of code fragments that are similar to each other.
- **A Clone Class** is a set of code fragments where any two fragments can form a clone pair.
- **A Clone Class Family** is a set of clone classes that belong to the same aggregation group, such as a package, directory, file, class, or function.

As mentioned, our stakeholders use Kam1n0 to find code clones, and Kam1n0 supports the detection of similar functions by finding block-to-block clone pairs. More details of Kam1n0 are given in the next section.

2.4 Kam1n0

Code clone analysis is an intensive and time-consuming process that aims to identify software behaviours, locate code clones, and recognize vulnerabilities. In large software systems, manually detecting and analyzing clones is particularly challenging, and identifying vulnerabilities in large code bases can be difficult because code cloning frequently occurs in software engineering. These challenges are increased when source code is unavailable and code clone analysis is performed on assembly code. Typically, analysts have to inspect many different applications, each of which can consist of millions of assembly-level instructions. The challenge of efficiently performing manual assembly code clone analysis motivated the creation of Kam1n0, an assembly code clone search engine [18].

2.4.1 Kam1n0 Search Process

Kam1n0 provides accurate, efficient, and scalable assembly code clone searching performance [18]. This tool is capable of finding cloned functions in either a single binary or across a set of binaries. In order to give a clear understanding of the Kam1n0 search process, I first provide terminology used in Kam1n0:

- **Target function:** An assembly function given as a query.
- **Target binary:** A binary file whose functions are given as queries.

- **Repository:** A repository that stores all indexed functions from binaries imported by a user.
- **Repository function:** an assembly function stored in the repository.
- **Clone search:** A process to search for similar functions in the repository when given a target function as a query.
- **Clone-search application**[1]: A term defined in Kam1n0. Kam1n0 allows a user to create *clone-search applications*. A *clone-search application* uses a certain search approach to identify clones. Different types of *clone-search applications* are discussed in the next section.
- **Clone search result:** A result of a *clone search*. All *clone-search applications* give the same format of results, and each result is saved in a JSON file that includes: 1) similarities between a target function or the functions from a target binary and the repository functions; and 2) the parameters of these functions and binaries. An example of a search result is attached in Appendix D.

The process of a single search in Kam1n0 is shown in Fig. 2.1. Kam1n0 allows a user to create one or multiple *clone-search applications*. In a *clone-search application*, the user can index one or multiple binaries imported from the user's computer to generate a repository. The user then can search a single target function (or a single binary that contains a list of target functions) with the repository functions. In reality, a user can build multiple *clone-search applications*, and run multiple searches in each *clone-search application*, producing multiple search results.

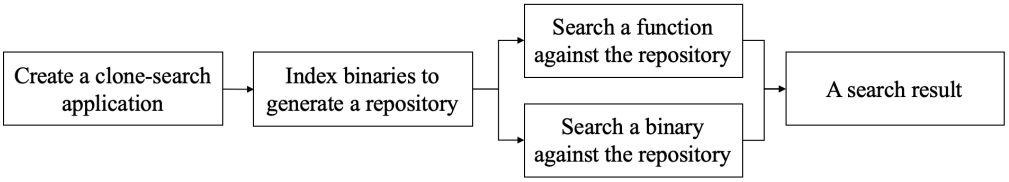


Figure 2.1: The process of a clone search in Kam1n0

[1]https://github.com/McGill-DMaS/Kam1n0-Community

2.4.2 Kam1n0 Clone-Search Applications

Kam1n0 provides three types of *clone-search applications*: Asm-Clone, Sym1n0, and Asm2Vec. The first two types of applications aim to identify *subgraph clones*, and the third one is used to identify semantic clones. This section introduces the three types of *clone-search applications*.

Subgraph Clone Search

Kam1n0 uses an IDA Pro2[2] disassembler to extract a list of assembly functions from binary files. Each assembly function is composed of several basic blocks. In Asm-Clone and Sym1n0 applications, by comparing the basic blocks from two functions, a list of *subgraph clones* can be found, where each *subgraph clone* is composed of a set of basic block clone pairs [18].

Assume we use an Asm-Clone or a Sym1n0 application to search a target function A with a repository function B. If one or multiple subgraph clones are found by comparing A to B, the similarity of A to B can be calculated by checking how many *subgraph clones* cover the graphs in A. On the contrary, if a user uses B as a target function and A as a repository function, the similarity of B to A can be different from the similarity of A to B because the graph size of A and B are generally different. Both Asm-Clone and Sym1n0 application search for subgraph clones and support Type I, II, and III clones, but they use different approaches. An Asm-Clone application only supports a single assembly language family, but a Sym1n0 application can search across different assembly language families.

Semantic Clone Search

In contrast to Asm-Clone and Sym1n0 applications, an Asm2Vec application aims to find lexical semantic relationships of two assembly functions [19], so this application supports Type IV code clones. In an Asm2Vec application, a representation learning model, also called Asm2Vec, is built to construct a feature vector for assembly functions [19].

Representation learning, also known as feature learning, is a term used in machine learning. The performance of machine learning methods highly depends on which data representations (i.e., features) are chosen [20]. Thus, researchers put efforts

[2]https://www.hex-rays.com/products/ida/index.shtml

into data pre-processing to gain a data representation that can improve machine learning efficiency [20], resulting in the field of representation (feature) engineering. However, representation engineering is a difficult and expensive manual process that requires an expert to have domain knowledge to create the features. In representation engineering, techniques are proposed to automatically discover representations.

By leveraging representation engineering, the Asm2Vec model does not require any prior knowledge about assembly code [19]. The only input required is a list of assembly functions used as queries. The following workflow is used for this model.

A neural network is built for the functions in the repository. After training the data in the neural network, the vector representations for each *repository function* are generated. Given a *target function*, the model can estimate the target function's vector representation. The similarity between a target function and a repository function is calculated by comparing the vectors of the target function to the vectors of the repository function.

Although Kam1n0 provides an efficient clone search technique for both textual and semantics clones using three types of *clone-search applications*, using the existing visualizations in Kam1n0 to analyze clones is still challenging, especially when we need to inspect a large number of clones. To address these challenges, I researched existing visualizations for code clone analysis, which are discussed in the next chapter.

Chapter 3

Related Work

The analysis of software can be broken down along a code abstraction spectrum based on the unit of analysis. At the low end of abstraction are snippets, lines of code, function blocks, etc., whereas methods, classes, files, modules, versions, programs, etc., are at the higher end of code abstraction. Accordingly, visualizations such as node-link diagrams, matrix-based views, or hybrids of the two views for code clone analysis are used to show different abstraction units [21]. In this chapter, I first discuss existing visualizations used for code clone analysis. Then, I describe matrix-based views for large-scale graphs, which provide insights on how visualizations can scale to the millions of clones possibly identified by Kam1n0.

3.1 Visualization for Code Clone Analysis

Node-link diagrams and matrix-based views are common ways to show code clones. In these visualizations, nodes often represent code abstractions such as software, packages, classes, and lines of code; links (also known as edges) represent relations between the code abstractions such as similarities and dependencies.

Node-link diagrams, including trees and networks, give high readability in small and sparse graphs. A matrix-based view gives a compact representation and performs well in large and dense graphs, but its readability is affected by the size and density of the graph [22][23][24]. To take advantage of both, researchers [25][26] also used hybrids of the two views to show code clones. In this section, we give some examples of these visualizations.

3.1.1 Node-link diagrams

In node-link diagrams, such as trees and networks, nodes are usually drawn as points or shapes (e.g., circles and rectangles) connected by links, indicating the relationships between items.

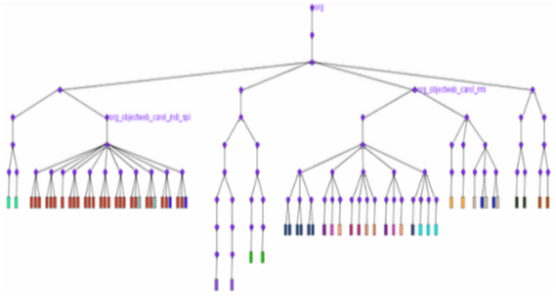
Trees

SoftGUESS [27] uses trees to show evolutionary code clone behaviors. The nodes in the trees can represent snippets, methods, classes, etc., and the edges can represent containment, dependencies, and cloning relationships. For instance, in Fig. 3.1a, circles represent packages, classes, and methods, and squares represent clone snippets. Squares with the same color indicate that the clone snippets are from the same clone class. The links between nodes illustrate the hierarchical containment of clone snippets in packages, classes, and methods. Different variations of trees are also applied to code clone analysis. For example, icicle plot graphs (see Fig. 3.1b) are used to show lines of code, but only two versions of software can be compared using this approach [28][29].

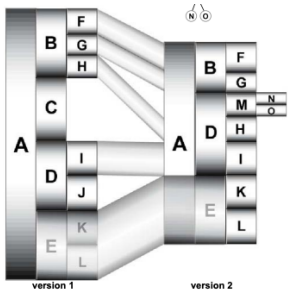
Hauptmann et al. [30] proposed a rudimentary approach using radial trees combined with edge bundle views to discover clones in software systems. Later, Hanjalić [31] applied this approach to a visual tool called ClonEvol, which shows changes across thousands of versions. Fig. 3.1c shows the visualization in ClonEvol. As we can see, the hierarchical structure of software, including directories, files, and classes, is illustrated as multiple-layer circles, where each circle represents a certain level in the hierarchy. The circle is divided into sections, and each section represents a node at that level. Connections between the sections show the clone relations of the nodes. By observing the changes in a series of radial trees, users can gain insights about how software evolves. However, each radial tree in ClonEvol can only show one version of a software system at a high level of abstraction, such as projects, files, and scopes. Showing a lower level of abstraction (e.g., functions) can greatly increase the size of the graph that needs to be displayed because a file or a project can contain a large number of functions.

In VisCad [9][32], a radial tree is also used to show the clone relationship between directories and files (see Fig. 3.1d). The hierarchy of directories and files in a software system is represented by nodes located along the concentric circles, and the centre red node is the root directory of the system. The diamond-shaped node represents the

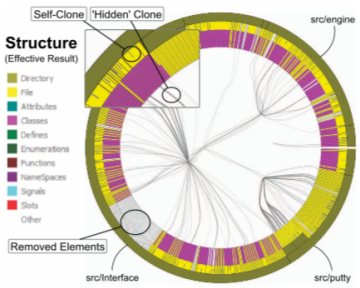
target directory, and other nodes represent directories that contain clones compared to the target directory. Directories and files that have no clone relations with the target directories are not shown in the graph.



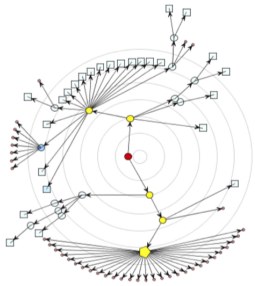
(a) Tree structure in SoftGUESS [27]



(b) Icicle plot graphs [29]



(c) A radial tree in ClonEvol [31]



(d) Hierarchical dependency graph in VisCad [9]

Figure 3.1: Tree-based diagrams for code clone analysis

Networks

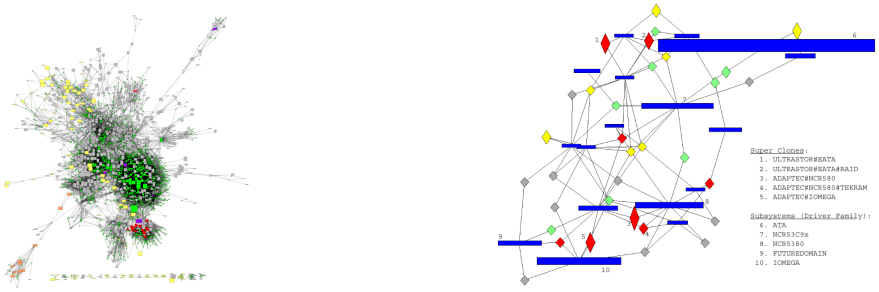
Networks are node-link diagrams without hierarchical relations between nodes—this type of visualization can also be used to show code clone relations. Similar to trees, a network contains nodes which represent projects, directories, files, etc. These nodes are connected with links that represent clone relations between nodes. Additional visual marks such as shapes, size, and colors can be included in the networks to provide more information. Some examples of networks used in code clone analysis are as follows.

Yoshimura and Mibe [33] conducted a case study, in which they applied a network to visualize code clones in a software system. In their study, circles represent files and links represent two files with a clone relation, where the size of a circle is scaled to the size of a file, and different colors represent different types of files (i.e., files with

clones are marked as red, and independent files are marked as grey). In addition, a set of clones can be grouped as a clone cluster represented by a green circle.

Besides circles, other shapes such as rectangles are also used to visualize code clones. Sæbjørnsen et al. [34] used connected colored boxes to show the files and clones found in a set of files in Windows XP (see Fig. 3.2a), where a green box with a larger size corresponds to a larger number of clones, and other colors indicate files from different sub-directories (e.g., the files in driver directories are shown as yellow boxes).

Jiang et al. [35] used a network view to show the clone cohesion and coupling between subsystems, including the clones within each subsystem and across multiple subsystems (see Fig. 3.2b). Instead of using a node to represent a single element (e.g., a file or a directory), they used a node to represent a clone class family. A rectangle node indicates clones from a single subsystem, and a diamond node indicates clones from multiple subsystems. Colors and sizes represent the number of clones in a node.



(a) Network shows clones in Windows XP[34] (b) Network with different shapes [35]

Figure 3.2: Network diagrams for code clone analysis

To summarize, node-link diagrams such as trees and networks are used to analyze code clones in many studies. However, a drawback of node-link diagrams is the scalability limit [36]. To address this issue, an alternative is to use matrix-based views.

3.1.2 Matrix-based views

Because of the need to present many-to-many comparisons, I chose to explore matrix-based views, which have been commonly used in code clone analysis.

An Adjacency Matrix (see Fig. 3.3) is a typical matrix-based view for a many-to-many comparative analysis. A node-link diagram described in the previous section

can be visually encoded as an Adjacency Matrix, where nodes are placed along the rows and columns of a square region, and the links between nodes can be shown by the visual representations (e.g., color, luminance) at the intersections (that is, the "cells") of rows and columns. Adjacency matrices are sometimes known as scatter plots or color heatmaps if colors are introduced, and we refer to all of these visualizations as matrix-based views in this thesis. Next, we describe how matrix-based views are used for code clone analysis.

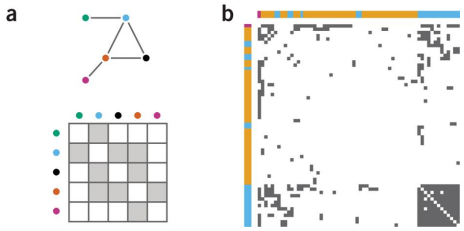
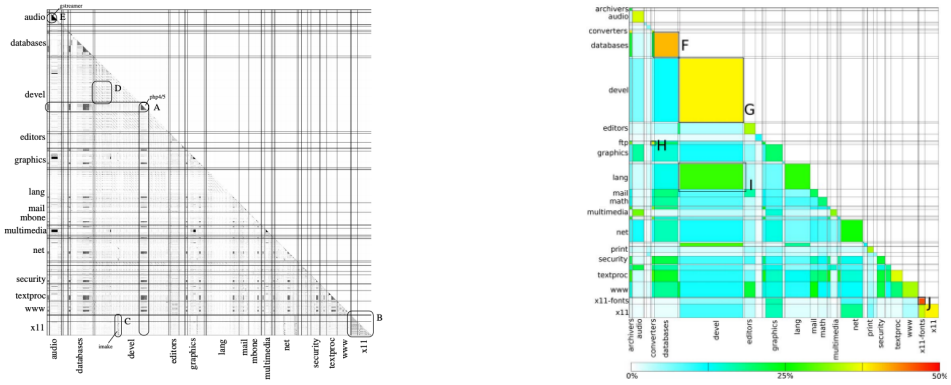


Figure 3.3: Adjacency Matrix [1]

CCFinder [37] is a code clone detection system, which shows the clone detection results in matrix-based views in order to compare file pairs and code fragment pairs. Later, a number of studies [2][27][32][35][38][39] focused on visualizations of clones detected by CCFinder. For example, Gemini [38] uses a matrix-based view to visualize CCFinder detection results and analyze code reuse in multiple programs, with a sorting algorithm applied to the basic matrix-based view. Additionally, Gemini is applied to show the clones in large-scale software systems [40].

Similarly, D-CCFinder [2] uses scatter plots (see Fig. 3.4a) and color heatmaps (see Fig. 3.4b) to show frequently used code. Both scatter plots and heatmaps are matrix-based views, and they can produce recognizable patterns. In D-CCFinder's visualizations, the rows and columns represent categories, where each category is a set of projects that share a specific feature. The intersection (i.e., a cell) of a row and a column shows the code clone coverage of two categories, where the code clone coverage is defined as the ratio of *the number of code clone fragments* to *the total number of lines of code from two categories*. Alternatively, the rows and columns can also represent projects, and a cell shows the code clone coverage of two projects. The difference between a scatter plot and a color heatmap is the visual representation in the cells. In a scatter plot, a dot is added into a cell if clones exist in the two categories or projects. In a heatmap, colors scaled by the code clone coverage are added to the cell. Similarly, Livieri et al. [41] also used a heatmap to display the code clone coverage ratio between different versions of the Linux kernel.



(a) Scatter plots (b) Color heatmaps

Figure 3.4: Matrix-based views used in D-CCFinder [2]

Matrix-based views can also show clones at different levels of abstraction. For instance, Cordy [42] implemented live scatter plots to analyze the similarities between two systems at multiple levels of data abstraction, such as files, directories, and subsystem directories. Asaduzzaman [9] used scatterplots to compare directories and files.

3.1.3 A Hybrid Approach

Combining a node-link diagram with a matrix-based view is an another approach to visualize code clones. Beck and Diehl [25] used an Adjacency Matrix with layered icicle plots attached to the sides to show the comparison of two software architectures. Similarly, Matrix Zoom [26] is a matrix-based view with hierarchy trees attached to the top and the left of the matrix. This way, users are able to see both the hierarchical architecture of a software system and the many-to-many comparisons between directories and files in that hierarchy at the same time. As both studies described above can handle large-scale graphs, a detailed introduction is given in the next section.

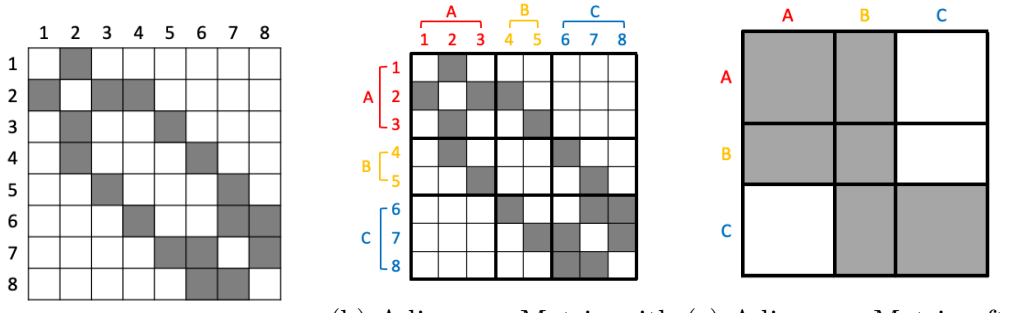
3.2 Matrix Views for Large-Scale Data

In the previous section, I discussed several visualizations used for code clone analysis, and I found that the matrix-based view might be a good option because it can show many-to-many comparisons. Adjacency matrices (see Fig. 3.3) may be a practical way to display large, undirected networks [1] as they can scale up to 1,000 nodes

and 1,000,000 links [36]. However, my initial research showed that the search results from Kam1n0 can consist of millions of nodes and links, which is where the Adjacency Matrix falls short. Luckily, many researchers have proposed some approaches for large matrix-based graphs. There are limits, however, to what humans can understand and what the computer can display with current visualization techniques. One way to address these issues for large-scale data is to visualize data by showing an overview first and then providing details on demand [43][44]. Some researchers have proposed zoomable Adjacency matrices to overcome the scalability limitation of an Adjacency Matrix.

3.2.1 Zoomable Adjacency Matrix

In a zoomable Adjacency Matrix, nodes are aggregated at a higher level of abstraction. Fig. 3.5a shows a typical Adjacency Matrix, as mentioned in the previous section. If nodes $1-8$ are aggregated as three nodes, A, B, and C (see Fig. 3.5b), an Adjacency Matrix can be used to show the comparison among the three nodes at a higher level of abstraction (see Fig. 3.5c). Additionally, the different levels of Adjacency matrices can be switched by zooming action, and this is called zoomable Adjacency matrices. Next, we discuss how zoomable Adjacency matrices are used for data with hierarchical and non-hierarchical relations.



(a) Adjacency Matrix

(b) Adjacency Matrix with aggregated nodes

(c) Adjacency Matrix after aggregation

Figure 3.5: Zooamble Adjacency Matrix

Hierarchical data

Some examples of using zoomable Adjacency matrices for hierarchical data such as software architectures, phone traffic, and medical data are described below.

Van Ham [45] hierarchically decomposed large systems into five levels of abstraction: system, layer, unit, module, and class. The parent nodes at a high level of abstraction in the hierarchy are shown first. After a user zooms in on an area, the child nodes at a low level of abstraction in the zoomed area are shown in the visualization.

Beck and Diehl [25] also used a zoomable Adjacency Matrix to analyze software architectures. They introduced a scalable analysis tool to compare two software architectures, and each software includes a hierarchical relation: packages, classes, and methods. An Adjacency Matrix view is used to compare the elements in the hierarchy, and layered icicle plots are attached to the sides of the matrix to show the hierarchy (this is also mentioned in the previous section). Zooming in on an element of the hierarchy can enlarge the rows or the columns, which enables the comparison of elements at lower levels of hierarchy. Besides layered icicle plots, different tree-structured visual representations can be used on the sides of a matrix. Matrix Zoom [26] is a zoomable Adjacency Matrix that attaches hierarchy trees to the top and the left of the matrix (also mentioned in the previous section). Matrix Zoom can also be used to show medical data, such as cancer information and phone traffic.

To summarize, in the zoomable Adjacency Matrix for hierarchical data, an overview displays parent nodes (i.e., nodes at higher levels in the hierarchy) at rows and columns, and the comparison of two parent nodes is shown in the intersection of a row and a column. By zooming in on a specific area, the comparison between the child nodes (i.e., nodes at lower levels of the hierarchy) of the specified parent nodes can be shown.

Non-hierarchical data

Zoomable Adjacency matrices can also be used for data without any hierarchical structure. In this case, aggregation techniques are applied to group the nodes, with each group containing multiple nodes. Two examples of zoomable Adjacency matrices showing non-hierarchical data are described as follows.

Van Ham et al. [46] introduced a technique to collapse multiple cells (i.e., intersections of rows and columns) into one cell, using color luminance to show the density of the links of the cells before collapsing. ZAME [47] is another zoomable Adjacency Matrix showing non-hierarchical data. ZAME provides eight choices of glyphs shown in the cells of the Adjacency Matrix (such as color shade, average, and histogram)

to show the distribution of the comparison between two aggregated nodes. These aggregation and zooming techniques can dramatically reduce the size of the graph to be displayed and allow the Adjacency Matrix to show millions of nodes and billions of edges.

In summary, this chapter gives a set of visualizations for code clone analysis, including node-link diagrams and matrix-based views. To deal with large-scale data, many researchers have applied aggregation and zooming techniques to matrix-based views. In the next chapter, I describe the research methodology used in this study.

Chapter 4

Research Methodology

As mentioned, the goal of this research was to design an interactive visual tool that could efficiently show assembly code clones from Kam1n0 to help users better understand the extent of clones in a software ecosystem and assist in tasks such as finding vulnerabilities. I used a design study methodology [7], an increasingly prominent approach used in problem-driven visualizations, to achieve my goal.

Over a period of eight months, my colleagues and I collaborated with Kam1n0's creators, researchers in the Data Mining and Security (DMaS) Lab at McGill University[1]. We also worked with some of Kam1n0's end users, a team of reverse engineers from Defence Research and Development Canada (DRDC). We had regular meetings with these stakeholders to identify their main problems. As we exposed their issues, we developed requirements for new visualizations that would scale to their needs, and iteratively designed, implemented, and refined the visualization tool, CloneCompass, based on their feedback. The process of improving CloneCompass also helped us gain further understanding about the nature of the problem faced by our stakeholders (as I discuss in Chapter 5). In the final stages of my design study, I conducted a preliminary evaluation and a Linux case study to evaluate CloneCompass.

This chapter gives an introduction of the design study used in this research. I summarize my four-stage design study, and describe what I did in each stage.

[1] http://dmas.lab.mcgill.ca/

4.1 Design Study

In my design study, real-world problems that domain experts face are addressed by appropriate visualization solutions [7]. As my colleagues and I cooperated closely with domain experts, we gained problem insights through ongoing discussions with our stakeholders. These problem insights guided us to refine and improve the visualization design.

The process of my design study can be summarized in four major stages, as shown in Fig. 4.1. A brief introduction of each stage is as follows:

1. **Knowledge Base:** we looked into the existing research knowledge base, such as foundations and methodologies that are applicable to this research. Throughout the study, we made contributions that were added to the existing knowledge base [48].

2. **Problem Characterization:** we identified and abstracted problems and requirements from our stakeholders through ongoing discussions with them.

3. **Development of Artifacts:** we designed and refined a viable artifact through the guidance of existing literature and according to the problems characterized in the first two stages.

4. **Evaluations:** we conducted a formative evaluation to gather ongoing feedback, and we conducted a preliminary evaluation and a case study to validate CloneCompass.

In a design study, two stages may overlap with each other, and one may also jump backward to prior stages to refine the previous understanding. Such overlapping of stages are referred as *iterative dynamics* by Sedlmair et al. [7]. The following sections introduce what we did at each stage and how the iterative dynamics occurred in this study.

4.1.1 Knowledge Base

Investigating the existing research *knowledge base* was the first stage in my design study. The main task at this stage was to learn theories and methods that are applicable to further stages. After my initial research of literature, I needed to go back to the first stage again, because we received new feedback from stakeholders,

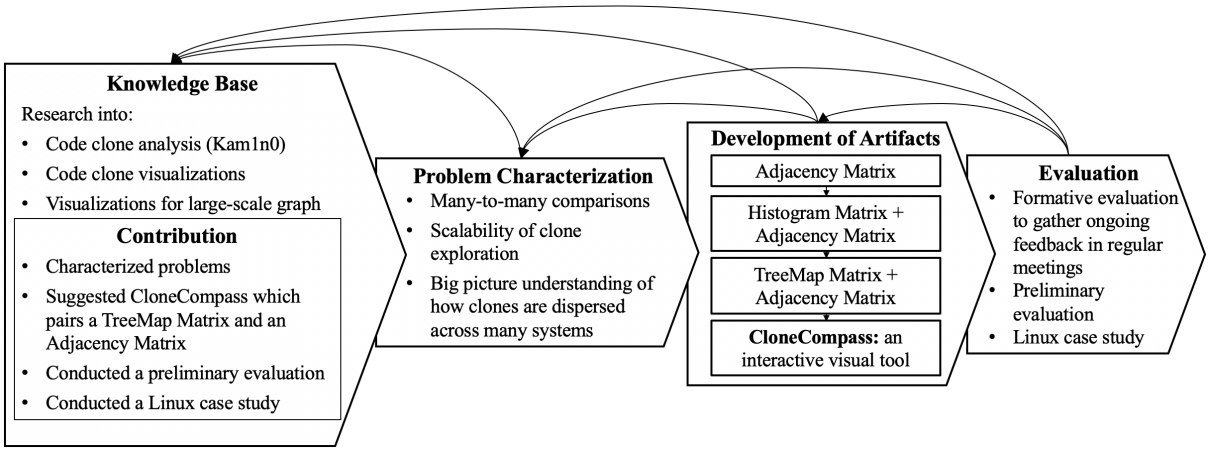


Figure 4.1: Design study research methodology applied in this thesis: the arrows above the blocks and overlaps between two stages imply the iterative dynamics of the stages

which required me to consider what I had not explored previously. As a result, this stage was influenced by the continuously updated information from the following phases.

At the beginning, I started with foundational code clone knowledge, such as *what code clones are* and *why it is important to analyze them*. Because our stakeholders use Kam1n0, I also investigated *what Kam1n0 was and its applicable scenarios*. The learning of code clones and Kam1n0 helped me understand our stakeholders' problems and what data and tasks can be abstracted to solve their problems. The foundational knowledge was described in Chapter 2.

Next, I explored the visualization literature because our stakeholders expected us to find a visual solution to understand code clones detected by Kam1n0. I investigated the visualizations for code clones first. However, I found out that most of the visualizations for code clones have scalability issues, so I sought visualizations for large-scale graphs. The existing visualizations formed the guidance for this work, suggesting into possible solutions to the problems we identified. Chapter 3 described the related visualization literature.

The knowledge base related to this work not only includes research into existing theories and methods, but also contributions that may be made during a design study [48]. Such contributions will be possibly useful in future research. In this study, the problems that we characterized may be faced by other code clone analysts, and our design artifact may be applied to further code clone analysis. More detailed

contributions are described in Chapter 10.

4.1.2 Problem Characterization

The second stage in my study was *problem characterization*, a process I followed to identify and abstract problems and requirements through ongoing discussions with stakeholders. This was a crucial phase as it allowed me to build a shared understanding between us and our stakeholders. At this stage, the stakeholders described the problems they had, and we distilled the problems. Problem characterization was also a cyclical process, where the problems that had already been identified were refined and new problems were posed based on continuous feedback from our stakeholders in the following stages.

Problem characterization impacted both the earlier and later stages. For the first stage (i.e., building knowledge base), the results of problem characterization may require updating our knowledge base. For the following stages, problem characterization was a fundamental element in developing the artifacts, which were expected to help stakeholders solve their problems. The problems we characterized are discussed in Chapter 6.

4.1.3 Development of Artifacts

The main stage of this study was the *development of visualization artifacts*. Together with my colleagues, I iteratively designed and refined visualizations with the guidance from the existing literature found in the first stage, the problems characterized in the second stage, and the ongoing feedback gathered from a formative evaluation in the last stage.

The problem characterization showed that we needed to visualize the assembly functions and their similarities. Initially, I implemented an Adjacency Matrix to show this information. However, when I used a dataset of over 90 million function-clone pairs (this dataset was provided by our stakeholders), the Adjacency Matrix could not load the data because of its scalability limits. To address this issue, I went back to the first stage and researched existing studies that focus on scalability optimization for an Adjacency Matrix.

As mentioned in Chapter 3, a practical solution to handle high scalability in a matrix-based view is to show an overview first, then focus on details on demand. Inspired by the studies of the zoomable Adjacency Matrix, I tried to use a “Histogram

Matrix" (see Fig. 4.2), a matrix-based view with cells represented by histograms to show the overview of a dataset, where each histogram shows the distribution of two aggregated nodes from the dataset. However, we discovered that the histogram wasted space on a size-limited screen. We then decided to use a TreeMap instead of a histogram in a cell: created a TreeMap Matrix to show an overview of the dataset, pairing an Adjacency Matrix to show the detailed information. We also designed and refined several features and interactions of CloneCompass based on the feedback from our stakeholders. Chapter 6 describes the details of the visual tool CloneCompass.

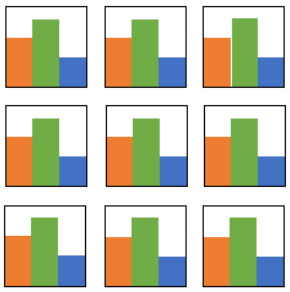


Figure 4.2: Concept of Histogram Matrix

4.1.4 Evaluation

The last stage of this study involved the evaluation of CloneCompass. Through ongoing meetings with our stakeholders, we carried out formative evaluations to gather feedback. We carefully considered their feedback and applied this essential information to our design. Such feedback helped us understand our stakeholders' needs and gave insights to refine our design. For instance, I implemented an animation to explain how the functions were grouped as aggregated nodes used in the TreeMap Matrix; however, our stakeholders stated that this animation might be useful in the documentation but not in the visualization, so I deleted this feature.

Besides formative evaluations, I also conducted a preliminary evaluation to validate CloneCompass. Two creators of Kam1n0 agreed to participate in the evaluation. During this preliminary evaluation, a "think-aloud protocol" [49] was applied to help me understand the participants' cognitive processes while the participants used CloneCompass, and to help me determine the usability of CloneCompass and its effectiveness at assisting with the analysis tasks. The process and results of the preliminary evaluation are discussed in Chapter 7.

At the end of this study, I carried out a Linux case study. In order to extend the applicability of CloneCompass, I used CloneCompass to explore similar files in the Linux kernel. Chapter 8 describes the process and findings of this case study.

Chapter 5

Problem Characterization

Kam1n0 helps our stakeholders efficiently search for assembly function clones across multiple binary files, and provides basic visualizations for users to analyze their search results. This chapter first describes the visualizations provided by Kam1n0, and then discusses the challenges that our stakeholders have when using them.

5.1 Existing visualizations used in Kam1n0

As mentioned in Chapter 2, Kam1n0 allows a user to first build a *clone-search application*, and then index multiple binaries in the *clone-search application* to generate a repository. Next, the user can either search for the clones of a single target function or the clones of multiple functions from a target binary file by comparing these target functions to the repository functions. Also, Kam1n0 allows the user to build multiple *clone-search applications*, and run multiple searches in each application. A single search result for target functions and their clones is visualized in three different ways: function subgraph views, detailed views, and summary views.

Function subgraph view

A flow graph view, a text diff view, and a clone group view[1] are used to compare the subgraphs of two functions. These views are useful for comparing the details of two functions when the users search for the clones of a given function. In this study, I did not focus on the function subgraphs but on a higher level of abstraction (e.g., functions and binaries).

[1] https://github.com/McGill-DMaS/Kam1n0-Community

Detailed view

A detailed view shows a single search result in a *clone-search application*, as can be seen in Fig. 5.1. A list of target functions are shown in Fig. 5.1 (a), in which the background color indicates which binaries each target function is from. After the user clicks on a target function in the list, a hierarchical tree in Fig. 5.1 (b) is shown. The first level of the tree shows the binary of the selected target function, the second level of the tree shows the target function itself, and the third level of the tree displays the clones of this functions ordered by similarity indicated by the green bar charts next to each clone function. In Fig. 5.1 (c), the centre node represents the same target function in Fig. 5.1 (b), and the nodes connected to the centre node indicate the clones of the target function. When a target binary contains a large number of functions, the target function list that is shown in Fig. 5.1 (a) can be very long. When a target function contains a large number of clones, the clone list in Fig. 5.1 (b) can also be very long. Exploring these long lists of target functions and clones can be very time consuming. Note that Fig. 5.1 only shows a single search result, and when a user runs multiple searches in multiple *clone-search applications*, it takes even longer to explore all the results.

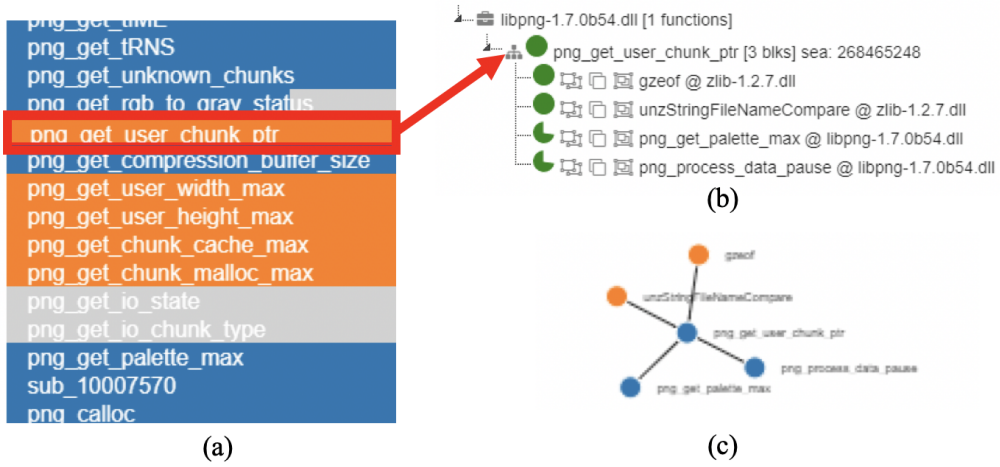


Figure 5.1: Detailed view in Kam1n0: (a) target functions list; (b) a detailed view shows a target function and its clones; (c) a node-edge graph view shows a target function and its clones

Summary view

A summary view in Kam1n0 (see Fig. 5.2) shows statistics about the clones of a target binary and the binaries in the repository. For example, in Fig. 5.2, the summary view on the right side shows the statistics about a target binary (the name of the target binary is not shown in a summary view) compared with a binary called *zlib-1.2.7.dll* in the repository. There are 17% of the clones with a similarity score more than m%, where m is the similarity threshold set by the user before each search. The clone similarity percentage is useful because it indicates how similar two binaries are, but if the user wants to know which clones exist in the two binaries, they need to use the detailed view shown in Fig. 5.1.

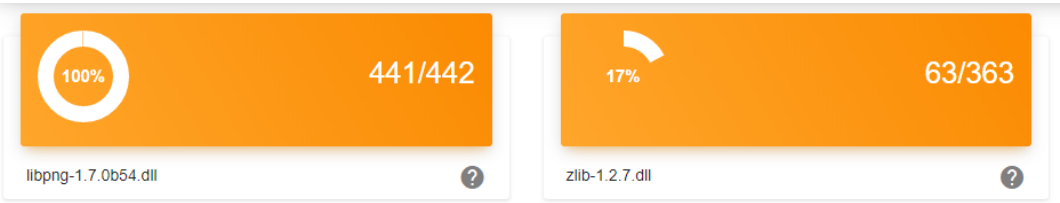


Figure 5.2: Summary view in Kam1n0

5.2 Visualization Challenges in Kam1n0

The previous section describes the initial visualizations provided by Kam1n0, but our stakeholders faced difficulties when using Kam1n0's detailed and summary views. Through interviews and ongoing discussions with our stakeholders and the refinement of the interim prototype visualizations, my colleagues and I discovered three key challenges that our stakeholders experienced when using Kam1n0's visualizations.

- **Many-to-many comparisons:** When using Kam1n0, reverse engineers can easily see the clones of one function (one-to-many comparison), but it is impossible to compare multiple functions or multiple binaries (many-to-many comparison) because a detailed view only contains a single target function and a summary view only contains a single target binary.

- **Scalability of clone exploration:** As mentioned in the previous chapter, it is difficult to efficiently explore clones because a search result from Kam1n0 can contain a very large number of assembly clones. These numbers are even larger

if the user builds multiple *clone-search applications* and runs multiple searches (potentially millions of function-clone pairs).

- **Big-picture understanding of how clones are dispersed across many systems:** When exploring clones in a software system or across software systems, reverse engineers are expected to have a big-picture understanding of the clones in the systems they consider. To support their analysis, they may need to answer a number of questions. For example, if they find a vulnerable code fragment in a part of a software system that has not changed in a very long time, clones of this fragment could have been spread to many other versions of the system and to other systems. Our stakeholders may also need to be able to identify the smallest and biggest clone in the system, and how many code clones occur across all versions of the software system.

The next chapter describes the final design of the new visualizations that may address these challenges.

Chapter 6

Solution Development: CloneCompass

To support the exploration of scalable many-to-many comparisons (the first two challenges I discussed in the previous chapter), I reviewed previous research concerning visualization techniques for large-scale data. I found that a possible solution is to show an overview first and then provide details on demand [43][44]. Building on this research, I propose CloneCompass, a tool that provides two visualizations, each of which targets a different level of abstraction: 1) a TreeMap Matrix at an aggregation level that shows an overview of the entire dataset of clones across one or more systems, and 2) a fine-grained view, an Adjacency Matrix, that shows further details of data points selected by the user. The user can then switch between these two visualizations to explore the whole dataset.

6.1 Visualization Design Process

As mentioned in Chapter 3, matrix-based views have been used to show comparisons between different software systems, versions, and files, etc. [2][41][42]. Inspired by these studies, I first tried an Adjacency Matrix to show the many-to-many comparison of functions. However, since the results from Kam1n0 may contain millions of function-clone pairs, a typical Adjacency Matrix is not able to show all the data because of its scalability limit. To address the scalability issues, previous studies [46][47][50] used aggregation techniques to reduce the amount of data shown. In an attribute-based aggregation technique, nodes in a graph are aggregated into groups

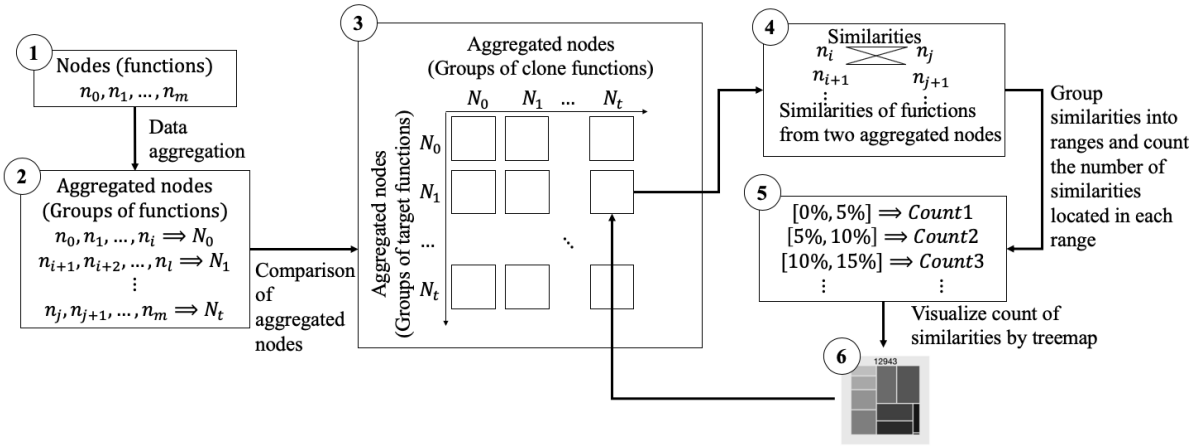


Figure 6.1: Design process of visualizations

by an attribute, and each group is treated as an aggregated node [51][52]. In our case, nodes are functions ($n_0, n_1, ..., n_m$ in Fig. 6.1 (1)) and they can be aggregated by one of their four attributes (i.e., binary, version, code size, and block size). These attributes can be extracted from Kam1n0 search results and match our stakeholder's requirements, such as finding smallest and biggest clones, analyzing clones across multiple systems and versions, identifying clones that have not changed in a very long time.

After aggregation, each aggregated node is a group of functions ($N_0, N_1, ..., N_t$ in Fig. 6.1 (2)). If we place aggregated nodes in the rows and columns in a matrix-based view (Fig. 6.1 (3)), each aggregated node in a row represents a group of target functions, and each aggregated node in a column represents a group of clone functions. In the intersection of a row and a column, a group of target functions and a group of clone functions are generated, and a list of similarities between target functions and clone functions is obtained (see Fig. 6.1 (4)).

Since the similarities can range from a very low value (two functions are different) to a very large value (two functions are very similar), splitting the similarities into distinct ranges can help the user focus on specific similarity ranges. I split the similarity ranges with a step of 5% and calculate the number of function-clone pairs having similarities located in each similarity range (see Fig. 6.1 (5)). For example, we may want to see a group of function-clone pairs with similarities from 75% to 80%.

The next section discusses which visual representation can be used to show the similarities of the functions from two aggregated nodes. After trying several visual representations, I decided to use the TreeMap as it performs well, both in expressive-

ness and effectiveness, and it saves space. As shown in Fig. 6.1 (6), a TreeMap is composed of a set of rectangles. In inner rectangles, the color luminance shows similarity ranges (i.e., a darker rectangle implies a higher similarity range); the size of an inner rectangle represents the count of function-clone pairs having similarities within a similarity range (i.e., a larger rectangle indicates a higher count). In an outer rectangle of a TreeMap, the number on the top shows the total number of function-clone pairs of the entire TreeMap.

6.2 TreeMap Matrix – An Overview

By placing the TreeMap at each intersection of a row and a column in Fig. 6.1 (3), a TreeMap Matrix is formed. From the TreeMap Matrix, users can see how the similarities of functions are distributed in the whole dataset. This section first gives examples of a TreeMap Matrix, and then describes the interactions and features included in a TreeMap Matrix.

6.2.1 Examples of TreeMap Matrices

As mentioned, the function groups in a TreeMap Matrix can be aggregated by one of their four attributes (i.e., binary, version, code size, and block size). Which attribute is used for the aggregation is set by the user. The examples of a TreeMap Matrix aggregated by different attributes are discussed below.

Ordered and aggregated by a quantitative attribute

Quantitative attributes of functions include code size and block size. After the functions are ordered by code size or block size, a binning process (described in Appendix A) is used to divide the ordered functions into distinct ranges, and the functions in each range are considered as a group.

Fig. 6.2 shows an example of a TreeMap Matrix ordered and aggregated by code size. The left and top nodes represent function groups with smaller code size. At a glance, we can see that the TreeMaps in the top-left area contain a large number of function-clone pairs because the numbers on the top of these TreeMaps are relatively large (e.g., the top-left corner TreeMap contains 61,483 function-clone pairs). Also, there are many large and dark rectangles in these TreeMaps, indicating there is a large portion of function-clone pairs with small code size and high similarities. In

contrast, the TreeMaps in the bottom-right area include a relatively small number of function-clone pairs and the rectangles with high similarities are very small, indicating this dataset contains a relatively small number of large clones.

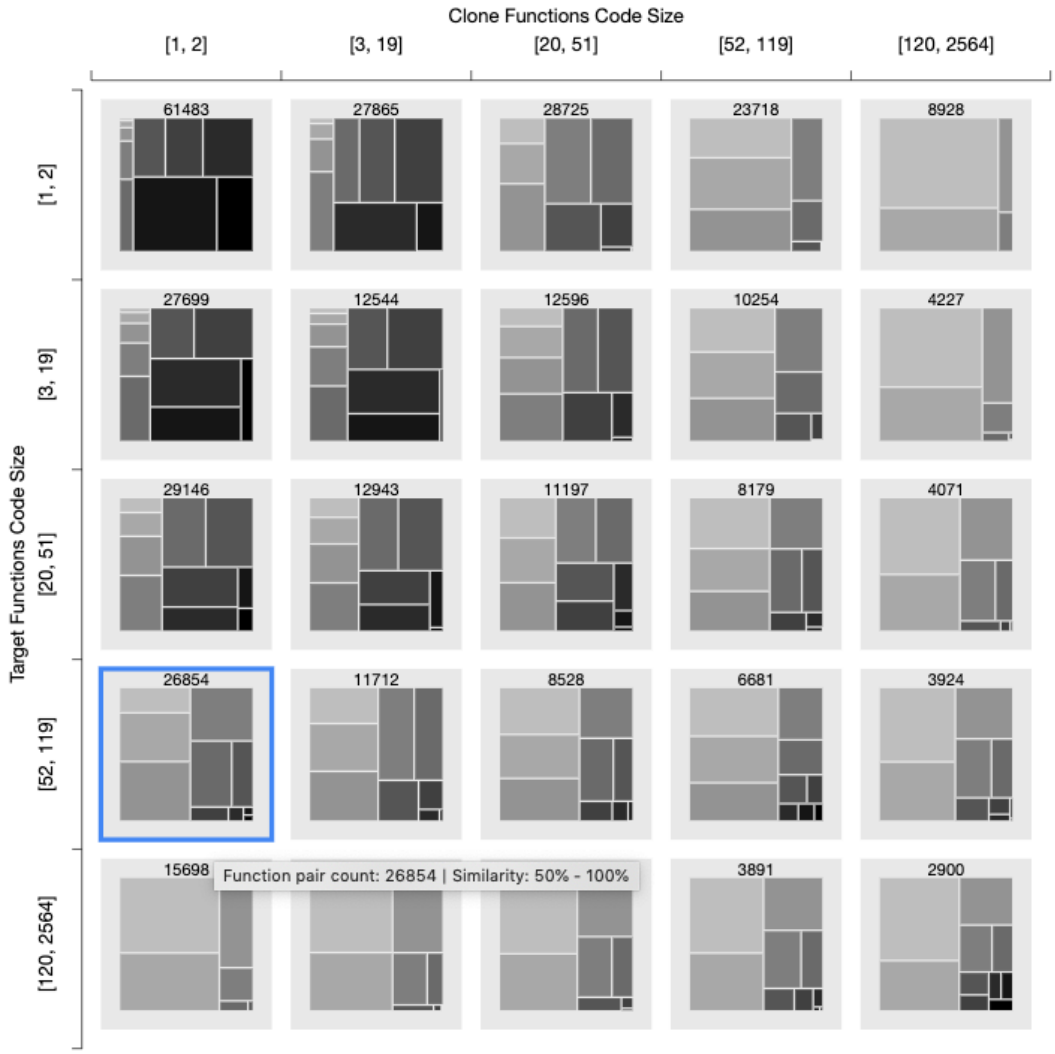


Figure 6.2: TreeMap Matrix ordered and aggregated by code size

Aggregated by a categorical attribute

For the categorical attributes (i.e., binary and version), functions from the same binary or version are aggregated into the same group.

Fig. 6.3 shows a TreeMap Matrix aggregated by version, where each TreeMap shows the similarity distribution of functions from two versions. All TreeMaps contain a similar number of function-clone pairs, and the similarity distributions in all the

TreeMaps are similar with each other. Since the dataset used in this example does not include the clones in a single software system, the TreeMaps along the diagonal are shown as greyed out squares with a number zero at the top of each TreeMap.

Similarly, a TreeMap Matrix can be used to compare multiple software systems, where each node represents a system, and each TreeMap shows the similarity distribuion of two systems.

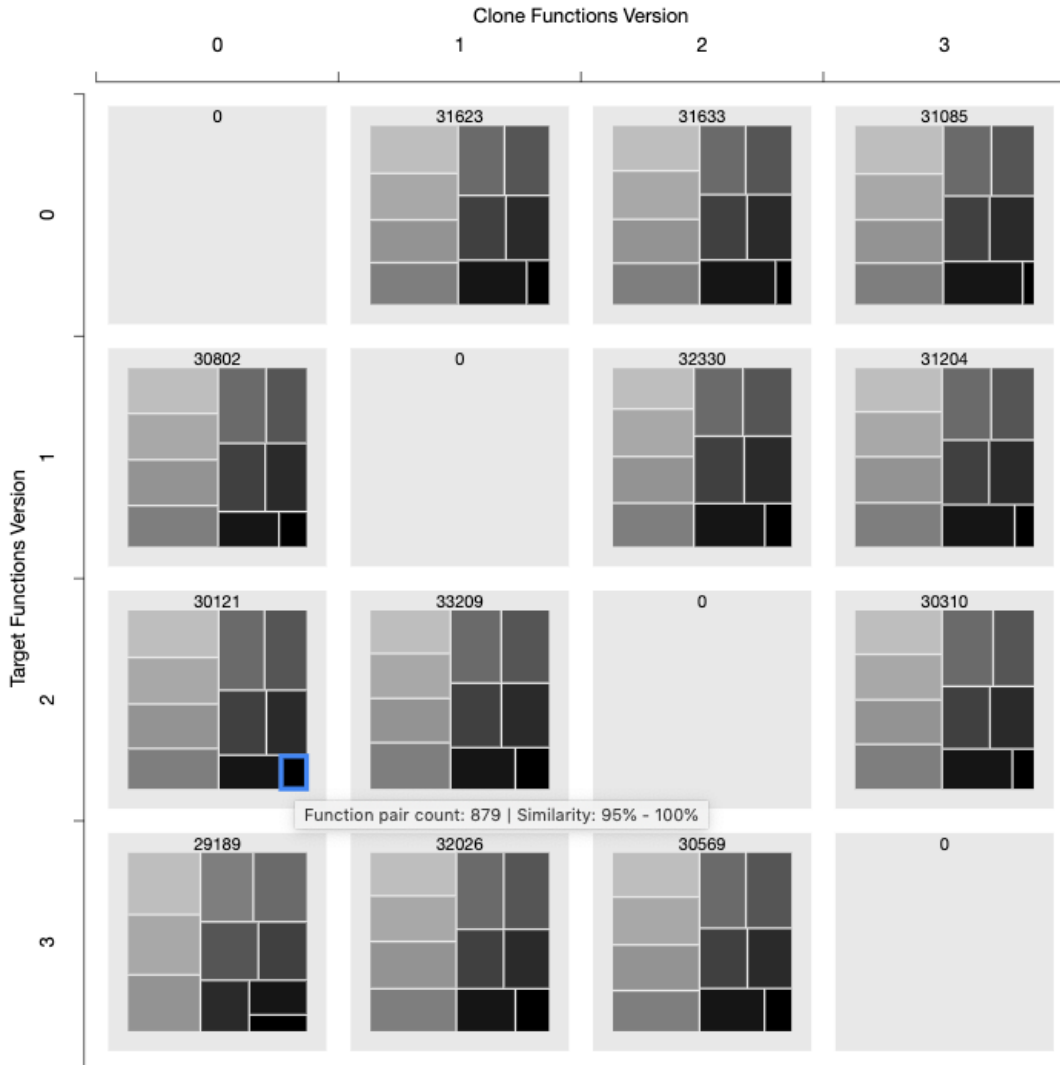
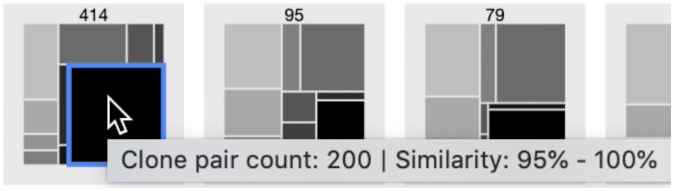


Figure 6.3: TreeMap Matrix aggregated by version

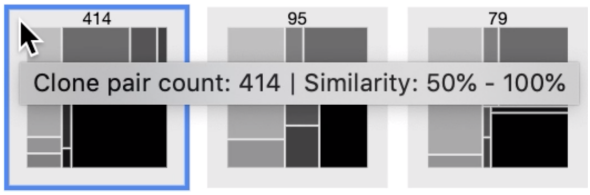
6.2.2 Tooltips in a TreeMap

I also implemented tooltips for the TreeMap visualization. If a user wants to see an accurate number of a collection of clones after observing the similarity distribution,

they can hover over either an inner rectangle (see Fig. 6.4a) or an outer rectangle (see Fig. 6.4b).



(a) Hover on an inner rectangle



(b) Hover on an outer rectangle

Figure 6.4: Hover and tooltip in TreeMap Matrix

To explain the tooltip in context, I use Fig. 6.2 and Fig. 6.3 from the previous section as examples. In Fig. 6.2, by hovering over the outer rectangle with the blue border, a tooltip box next to the TreeMap shows 26,854 function-clone pairs with similarities in the range of 50% to 100%, where target functions have a code size of 52 to 119, and clone functions have a code size of 1 to 2. Similarly, the hovered rectangle with the blue border in Fig. 6.3 contains 879 function-clone pairs with similarities from 95% to 100%, where the target functions are from version 2, and the clone functions are from version 0.

6.2.3 The use of pagination

To maintain readability when working with a large number of TreeMaps, pagination is used (on the right and at the bottom of the TreeMap Matrix) as shown in Fig. 6.5. Because Kam1n0 can produce a large number of results, the number of TreeMaps that can be shown is limited, especially when working with smaller displays. For example, I calculated the Kam1n0 search results of clones from several browsers which included more than one hundred aggregated nodes. This means that the number of TreeMaps could exceed ten thousand. In this case, showing all the TreeMaps at once is not feasible because they would be too small to read—pagination overcomes the screen size limitations and ensures all TreeMaps are readable.

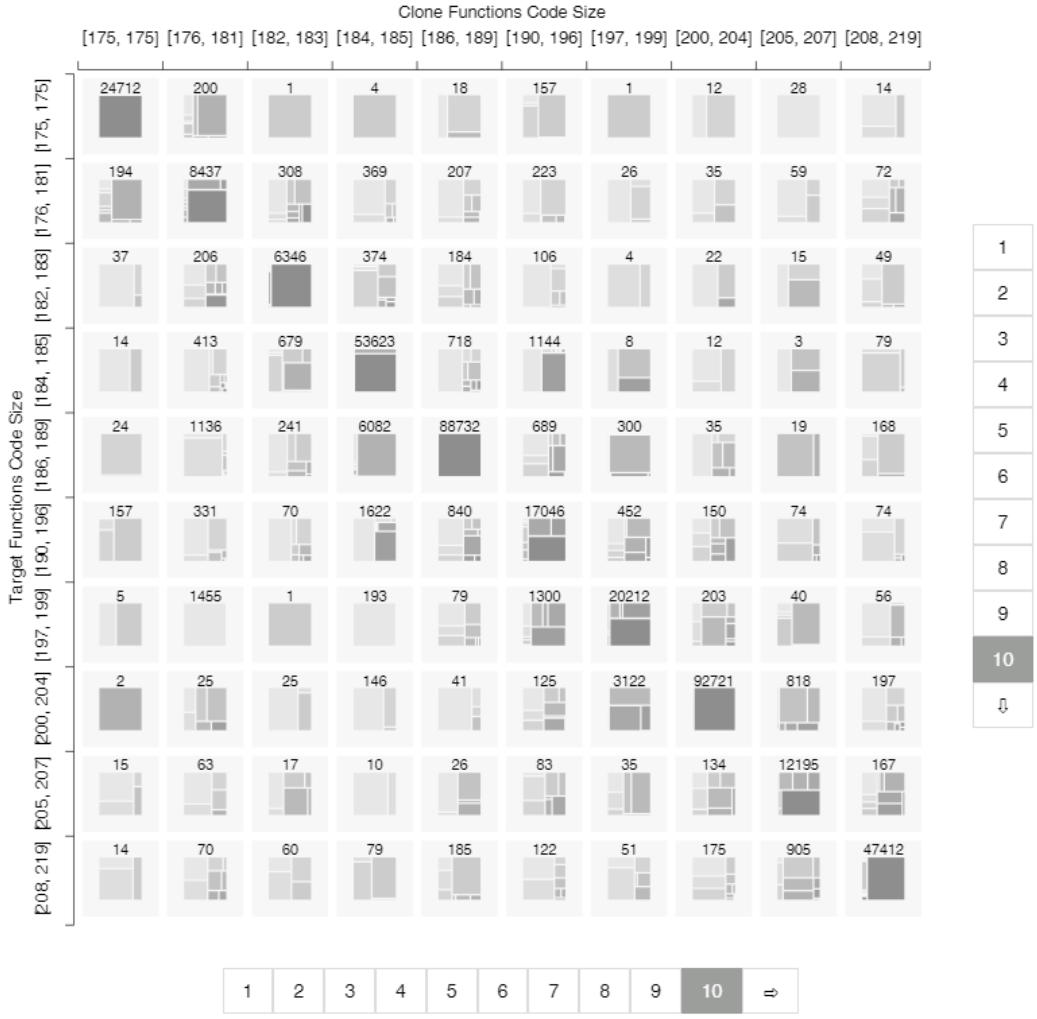


Figure 6.5: TreeMap Matrix with pagination

6.2.4 Multiple selections

The TreeMap Matrix shows an overview, but it does not provide detailed information such as the comparison between two specific functions. To provide details, previous studies [25][26][45] used zooming techniques to switch between overview and detailed views. However, zooming forces the user to focus on the data in a limited area of the screen—only the data from this area can be shown in a detailed view, and details of data displayed in another part of the screen cannot be inspected at the same time. In our case, when the user wants to see all data with high similarities (which are shown as dark rectangles in TreeMaps), zooming falls short because the dark rectangles are dispersed across the screen and it is not possible to zoom into these dispersed areas

at the same time. Similarly, if functions are ordered and aggregated by code size, zooming does not allow the user to observe clones of functions with a large code size because the corresponding TreeMaps are distributed across the bottom of the screen. Therefore, instead of zooming, multiple selections are used to allow users to see the detailed view of the data from any area: the user can select multiple outer and inner rectangles for a given TreeMap (see Fig. 6.6). By clicking on an outer rectangle, all functions from the two aggregated nodes of the TreeMap can be selected. By clicking on an inner rectangle, the functions from the two aggregated nodes of the TreeMap with *specified similarity* ranges can be selected.

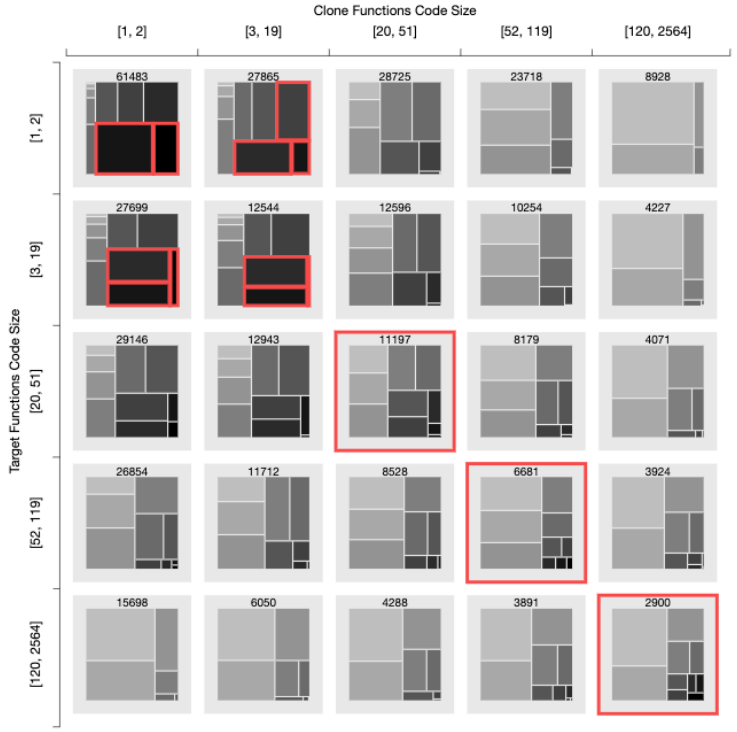


Figure 6.6: Multiple selections of inner and outer rectangles in TreeMap Matrix

6.3 Adjacency Matrix – A Detailed View

After rectangles are selected in the TreeMap Matrix, a comparison of functions from these rectangles can be shown in an Adjacency Matrix view. In the Adjacency Matrix, the rows and columns consist of target and clone functions ordered by an attribute, which is the same as the one used to aggregate the TreeMap Matrix (i.e., code size, block size, binary, or version). The intersection of a row and a column represents

the similarity between two functions. The Adjacency Matrix provides the following design features to assist the user.

6.3.1 Zooming and Panning

The number of function-clone pairs shown in the Adjacency Matrix varies by the selected rectangles. To ensure the user can still see details in this view when the number of function-clone pairs is large, they can zoom in, zoom out, and pan the visualization. The Adjacency Matrix before and after zooming and panning is shown in Fig. 6.7. When the square is big enough, the names of target and clone functions are rendered on the bottom and right side of the view. When hovering over a square, a tooltip shows the function names and similarity, and a "Hovered Function-Clone Pair" section (see Fig. 6.8) next to the Adjacency Matrix shows more detailed information including what binaries the functions are from.

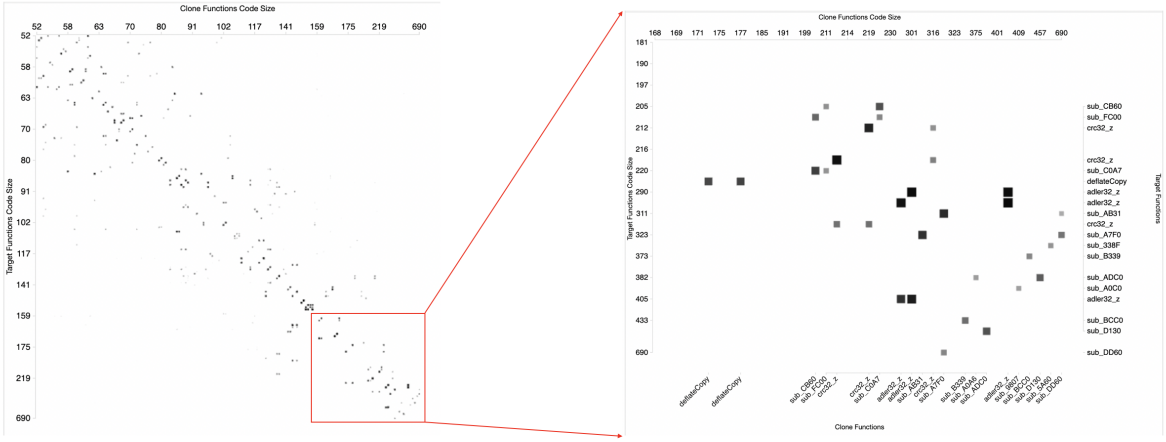


Figure 6.7: Adjacency Matrix before and after zooming/panning

6.3.2 Glyphs and Color Schemes

Different types of glyphs and color schemes are provided to ensure the Adjacency Matrix can adapt to various quantities of function-clone pairs. Users can bring varying size datasets and choose appropriate or preferred glyphs and colors.

Two glyphs, square (Fig. 6.7) and FatFont [53](Fig. 6.9), are used to represent the similarity in each cell of the Adjacency Matrix. Chang et al. [54] conducted an experiment to compare the effectiveness of different glyph usage in Adjacency

Hovered Function-Clone Pair	
Target Function	sub_B339
Clone Function	sub_BCC0
Similarity	0.9191467869938684
Target Binary	libz.so.1.2.11-gcc-g-O1-m32-fno-pic.bin
Clone Binary	libz.so.1.2.11-gcc-g-O2-m32-fno-pic.bin

Figure 6.8: "Hovered Function-Clone Pair" section in an Adjacency Matrix

Matrices with weighted edges. The authors showed that Square and FatFonts are effective choices for both detailed and overview tasks: FatFont can directly show the similarity value, but can be too small to be seen with too many simultaneous datapoints shown in the Adjacency Matrix, which is when the square glyph is more useful.

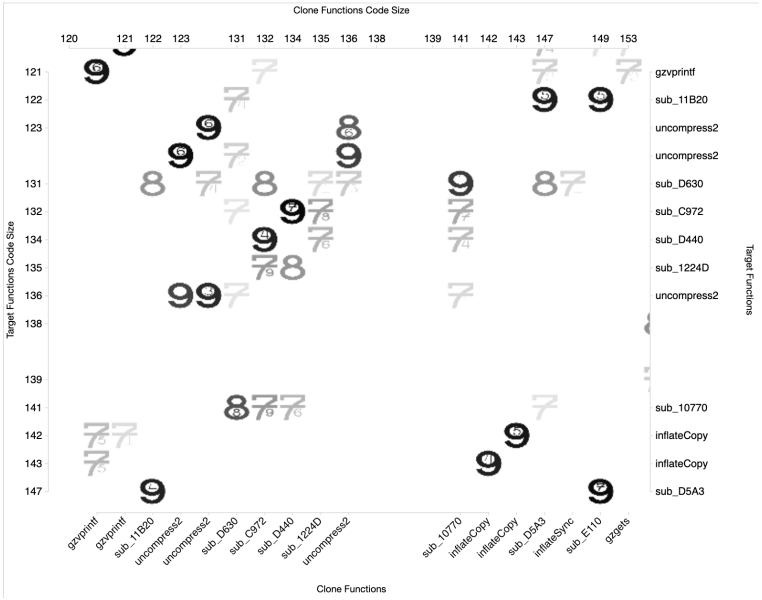


Figure 6.9: FatFont in Adjacency Matrix

When using square glyphs, different color schemes (shown in Fig. 6.10) can be selected. The default scheme is greyscale, in which both glyph luminance and size represent similarity, e.g., a large dark square represents high similarity. However, after I applied the Adjacency Matrix to large-scale datasets, I found that squares are not obvious when the similarity is low because squares become too light and small to be seen. Therefore, two other color schemes are provided without changing glyph

size: a diverging color scheme, BrBG[1], and a categorical color scheme, Viridis[1]. The Adjacency Matrices used difference color schemes are shown in Appendix B.

Different color schemes are useful in different situations: high similarities are obvious with greyscale, BrBG is useful when the user wants to see both low and high similarities, and Viridis distinguishes similarities via different colors (shown in Appendix B).

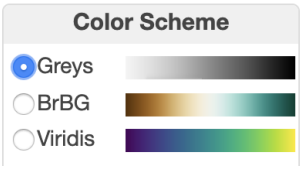


Figure 6.10: Color scheme choices and filters in Adjacency Matrix

6.3.3 Filters

Functions in the Adjacency Matrix are ordered by the aggregation attribute set by the user. Filtering allows the user to take attributes other than the aggregation attribute into account. By using filters (shown in Fig. 6.11), users can see multiple function-clone pairs constrained by different conditions. For example, when function-clones pairs with large code size and high similarities are shown in the Adjacency Matrix, functions from specific binaries can be shown by adding a binary filter.

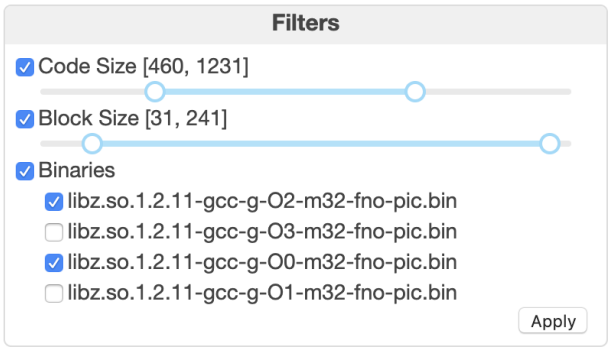


Figure 6.11: Filters in Adjacency Matrix

[1]https://github.com/d3/d3-scale-chromatic

6.4 Workflow

The previous section described the visualization design, i.e., the TreeMap Matrix and the Adjacency Matrix. This section describes the user's workflow when using CloneCompass.

CloneCompass is implemented on a visualization development platform, Lodestone,[2] that allows users to compose multiple panels of information. In each panel, the user can first build a TreeMap Matrix, and then switch between the TreeMap Matrix and an Adjacency Matrix. The workflow of CloneCompass is as follows:

1. **Pre-process data:** The user inputs Kam1n0 search results into a pre-processed script to calculate the data needed in the TreeMap Matrix.

2. **Load dataset:** The user then inputs pre-processed results into CloneCompass.

3. **Create panel:** The user creates a new visualization panel in CloneCompass.

4. **User configuration:** In the newly created panel, a similarity threshold and an ordering attribute are configured by the user. In a TreeMap Matrix and an Adjacency Matrix, only function-clone pairs with a similarity larger than the similarity threshold are included. The ordering attribute indicates the aggregation attribute used in the TreeMap Matrix and the ordering of functions shown in the Adjacency Matrix. If the user wants to analyze clones according to the size, they may choose the code size or the block size as the ordering attribute. If the user wants to compare clones across multiple systems or versions, they may choose the binary or the version as the ordering attribute.

5. **Observations:** After the configuration is set, a TreeMap Matrix is shown. By selecting multiple rectangles and clicking a switch button, the user can switch to an Adjacency Matrix that shows the functions from selected rectangles. After observing comparisons between functions, the user can return to the TreeMap Matrix. The user can explore the whole dataset by switching between these two matrix-based views.

[2]https://thechiselgroup.org/lodestone/

6.5 Implementation

CloneCompass is implemented on a visualization development platform, Lodestone,[3] that is built in Typescript[4] and uses React.js,[5] Pixi.js,[6] and D3.js.[7] Pixi.js is used to ensure the performance of the large-scale data displayed in the Adjacency Matrix, and D3.js enables visualization algorithms to operate smoothly. SQLite3[8] is used to save pre-process results because SQLite3 is a file-based dataset engine that is easily transferred to users. We deploy Lodestone as an Electron[9] app to our participants along with the SQLite database.

The performance of CloneCompass is measured on an Intel Core i7 CPU @ 3.40 GHz computer with 16.0GB of RAM and an NVIDIA Quadro 600 graphics card. Two datasets are used in the measurement: a small dataset contains a collection of the zlib libraries compiled with different optimization techniques; a large dataset contains a collection of browser binaries including Chromium, Opera, and Firefox, SRWare Iron, Comodo Dragon. The small dataset includes 532,730 function-clone pairs and 899 functions. The pre-processing takes less than one minute. There is no noticeable latency for loading and interactions. The large dataset includes 94,082,393 function-clone pairs and 873,626 functions. The pre-processing takes about 2.5 hours. The loading time of a TreeMap Matrix is 8 seconds. However, the loading time of an Adjacency Matrix depends on the number of function-clone pairs selected in the TreeMap Matrix. For example, it took less than three seconds to load 7486 function-clone pairs and 6 seconds to load 20,212 function-clone pairs in the Adjacency Matrix.

[3] https://thechiselgroup.org/lodestone/
[4] https://www.typescriptlang.org/
[5] https://reactjs.org/
[6] http://www.pixijs.com/
[7] https://d3js.org/
[8] https://www.sqlite.org/index.html
[9] https://electronjs.org/

Chapter 7

Preliminary Evaluation of CloneCompass

As part of my design study, I conducted a preliminary evaluation with our stakeholders using CloneCompass. My approach was to ask them to use CloneCompass to explore the clones generated by Kam1n0. During the exploration, I asked them to use a "think-aloud protocol" [49], where the participants were encouraged to verbalize their thoughts while using CloneCompass.

Rather than specifying user tasks for this evaluation, I asked the participants to list questions they might ask about clones and vulnerabilities before using CloneCompass. This enabled me to identify: 1) if the challenges captured previously (described in Chapter 5) truly exist when stakeholders explore the search results from Kam1n0; 2) potential challenges we had not identified earlier in our work.

7.1 Datasets and Participants

The evaluation was performed by two of Kam1n0's creators. I provided the participants with two datasets including: 1) a collection of zlib libraries compiled with different optimization techniques, in which Kam1n0 found 532,730 function-clone pairs; and 2) a larger dataset of a collection of browser binaries (including Chromium, Opera, Firefox, SRWare Iron, Comodo Dragon) and two other binaries (a libpng and a zlib library), in which Kam1n0 found 94,082,393 function-clone pairs. The first dataset was generated based on search results of demo binaries from Kam1n0's

GitHub repository[1]. The second dataset was provided to us by our stakeholders and helped demonstrate the scalability of our approach. Each of these different datasets are interesting to explore in terms of clones and potential vulnerabilities. Additionally, I encouraged the participants to load their own datasets. As a result, one participant used the two datasets I provided, and the other participant used their own dataset of binaries that contain known vulnerabilities.

7.2 Procedure

Tool exploration was remotely observed and recorded via screen sharing during a video conference, with participants talking through their thought and exploration processes and asking questions whenever they arose. The steps of the evaluation were as follows:

- Preparation:
    1. As mentioned in the previous section, one participant used their own datasets, and the other participant used the provided datasets. For the participant that used their own data (search results from Kam1n0), the participant ran a data pre-processing script before the interview. This script populated an SQLite3 database with consumable results usable within CloneCompass. For the other participant that used the data provided by me, this step was skipped as I sent him the SQLite3 database.
    2. I asked both participants to list questions that they expected to answer using CloneCompass. For example, how to find clones and vulnerabilities from multiple binaries, how to identify vulnerability clusters, and what is the relationship between similarities and function attributes.
- Remote observations (1 hour each):
    1. I conducted a simple overview training session to familiarize the participants with the layout of CloneCompass and its essential features.
    2. The participants explored CloneCompass, verbalizing their thoughts processes and asking questions when necessary. I interfered minimally and assisted if progress halted or if questions were asked.

[1] https://github.com/McGill-DMaS/Kam1n0-Community

- I conducted a feedback session where I asked both participants questions about CloneCompass: e.g., what information can and cannot be found in the two matrix-based views, and what problems can and cannot be solved using CloneCompass. I also asked user experience questions: e.g., are the two matrix-based views easy to read and use, and is CloneCompass user friendly. The questions asked during this feedback session are attached in Appendix C.

7.3 Findings

Based on participant descriptions during the exploration of CloneCompass and the feedback they provided, I summarized the findings confirming the three challenges identified previously: *many-to-many comparison*, *scalability of clone exploration*, and *big picture understanding of how clones are dispersed across many system*. Also, the evaluation results allowed me to analyze the participants' user experience and enabled me to refine the user requirements.

7.3.1 Support for Many-to-Many Comparisons

I found that both the TreeMap Matrix and the Adjacency Matrix performed well for many-to-many comparisons. For example, the two participants first used the TreeMap Matrix to see many-to-many comparisons between versions, and then chose two versions to perform a detailed comparison. With many-to-many comparisons in the Adjacency Matrix, the two participants were able to compare multiple functions and see clusters of clones. For example, by loading multiple vulnerable binaries in CloneCompass, the participant who used their own data identified clusters of vulnerabilities. By exploring these clusters, the participant could then identify potentially undiscovered vulnerabilities that had been grouped alongside known vulnerabilities.

7.3.2 Support for Scalability of Clone Exploration

I originally proposed the TreeMap Matrix view to overcome the scalability limit of Adjacency Matrices. Based on my observations, a user can efficiently explore a dataset with over 90 million function-clone pairs by switching between the TreeMap Matrix and the Adjacency Matrix. This is beneficial when doing large clone analyses. For example, one participant stated that for large binary files, if we used editors tools to

analyze the clones, these tools could take two minutes to load the binary files, but it did not happen in this tool.

However, the TreeMap Matrix displays a limited number of TreeMaps. For example, when the participant who used their own data analyzed clones, the dataset consisted of 762 binaries and 4,000 function-clone pairs. When attempting to show functions aggregated by binaries, the TreeMap Matrix did not load. Although the number of function-clone pairs was quite low in this dataset, the number of TreeMaps was 762*762, which is far higher than the number of TreeMaps aggregated by code size (108*108) in the dataset with 90 million function-clone pairs.

7.3.3 To Provide a Big Picture Understanding of How Clones Are Dispersed Across Many Systems

In terms of this challenge, I found that both participants obtained useful information to help them understand the analyzed systems. By using a TreeMap Matrix, one participant said they could observe *"the distribution of the clones with respect to certain factors such as block size, code size, versions, etc"*. Also, CloneCompass helped the user find *"the small/large clones of large functions"* and *"compare functions with large block size to functions with relatively small block size to check if there are any surprises"*. One participant suggested that CloneCompass could help users *"get an overview and have a general feeling of what the binaries and malware do"*, and it was suitable for users who *"do not have enough knowledge of the binaries in hand"*. Overall, both participants found the following information useful in their clone analysis by using CloneCompass: 1) clusters of clones and vulnerabilities; 2) new vulnerable functions that are similar to the existing vulnerable ones; 3) clones that have similarity within a specific range; 4) clones of multiple binaries; and 5) how similar a set of clones are compared with other clones.

I also discovered a new question that was not previously identified but that the participants found answers to by using CloneCompass. One participant created two panels and placed them side by side. One panel showed the TreeMap Matrix aggregated by code size, and the other one showed the TreeMap Matrix aggregated by block size. By comparing the two panels, the participant found many dark rectangles (i.e., clones with high similarities) in the TreeMaps aligned on the diagonal in the TreeMap Matrix aggregated by block size. However, in the TreeMap Matrix aggregated by code size, the TreeMaps along the diagonal contained more light rectangles

(i.e., clones with low similarities). According to the distribution of the diagonal, the participant concluded that highly similar clones shared similar block sizes rather than similar code sizes in this dataset—in other words, clones tend to have similar block sizes.

7.3.4 User Experience

A positive user experience was confirmed through my observations of the participants using CloneCompass and the questions asked in the interviews. CloneCompass was seen as *"user friendly"*, and both participants did not have many difficulties using the tool (according to their feedback). However, since the visualizations contain a lot of information, it is possible that *"for the first time users, they may not get it right away — they may need time to explore around and understand the meaning behind, but if you see a demo that would be sufficient"*. I summarize the positive and negative user experience points raised below.

TreeMap Matrix

Both participants reported that the TreeMap Matrix was easy to read and understand, but their initial understanding of the number at the top of each TreeMap would benefit from a tooltip. One participant particularly liked the multiple selection functionality. However, I also found a major limitation with the pagination feature: one participant thought it was hard to use because *"I do not know which combination of pages I should try"*.

Adjacency Matrix

Both participants reported that the Adjacency Matrix was easy to read and understand. However, the information in the tooltip of a cell only includes function names and similarities, which, according to participants' feedback, was not detailed enough—it would be better if other information was added, such as parent binary. Also, I received feedback on the glyphs used in the Adjacency Matrix: one participant remarked that *"FatFont is easy to use because it can see the actual value"*.

Interacting Across Two Matrix-based Views

CloneCompass allows a user to switch between two matrix-based views, through which the user can explore the whole dataset. However, when switching between two views, one participant was concerned because they forgot what had been already selected in the TreeMap Matrix. This is a meaningful insight: an annotation technique or a browsing history would help users remember what data has been selected in the TreeMap Matrix. Since the similarity range of an Adjacency Matrix is changed by the selected rectangles in a TreeMap Matrix, to differentiate the low and high similarities in an Adjacency Matrix, the color of squares is scaled by the similarity range. However, one participant suggested the use of a consistent color to represent a similarity, which indicates that giving the user options to scale the colors might be necessary.

7.3.5 Refining User Requirements

The results of the preliminary evaluation also helped us refine the requirements that we previously identified. This section focuses on these refined requirements.

Questions identified in the big picture understanding of clones

In this evaluation, both participants found that many-to-many comparisons and scalable clone exploration were satisfied by the proposed solution. Our interpretation of providing a big picture understanding of how clones are dispersed was correct because both participants expected to see the distribution of clones. However, the questions that we identified and prioritized, such as *finding smallest/biggest clones*, were not the participants' focus.

User-defined aggregation attribute

As mentioned, a TreeMap Matrix can be aggregated by four attributes: code size, block size, binary, and version. The first three attributes are saved in the search results from Kam1n0. However, versions are not saved in the search result, and versions need to be extracted from the binary names in the pre-processing step using a regular expression defined by the user.

Inspired by the fact that versions are extracted from binary names according to a regular expression, one participant suggested that a user-defined aggregation attribute could be introduced in CloneCompass to compare the clones based on this

new attribute. During the pre-processing, a user could write a regular expression defining how to extract any attributes from the dataset. For example, compiler names are sometimes included in the binary names, and a regular expression could extract compiler names enabling the TreeMap Matrix to compare clones from different compilers. Similarly, the user could create any attributes to aggregate the TreeMap Matrix by defining a regular expression to extract the needed attributes from the existing attributes.

Interactions between CloneCompass and Kam1n0

One participant reported that efficient interactions between Kam1n0 and CloneCompass could be useful for their workflow. For example, when a user clicks a square in the Adjacency Matrix, CloneCompass could jump to a location in the code within Kam1n0. This might help a user perform more detailed and exploratory analyses of clones.

To summarize, this chapter describes the preliminary evaluation I conducted with our stakeholders. The result showed that CloneCompass could visualize the assembly clones identified by Kam1n0. Since Kam1n0 is a specialized tool designed for assembly code, I wished to see if CloneCompass could also be applied to other use cases. Therefore, I conducted a Linux case to extend the applicability of CloneCompass, which is discussed in the next chapter.

Chapter 8

Linux Case Study

To extend the applicability of CloneCompass, I applied CloneCompass to visualize clones in the Linux kernel. Both the TreeMap Matrix and the Adjacency Matrix were used to compare files in Linux, and four attributes were used to aggregate the files: 1) file size, 2) number of contributors, 3) year of each file created, and 4) the number of commits. This chapter describes the data preparation step, and then presents the findings after exploring Linux clone data using CloneCompass.

8.1 Data Preparation

The clones in the Linux kernel were identified by a commonly used clone detection tool: CCFinder [37] (specifically, a redesigned CCFinder: CCFinderX[1]). CCFinder can efficiently extract code clones based on a token-by-token comparison. I used CCFinder to compare the files in the Linux kernel v5.0.9[2], and then I processed the result as an SQLite dataset, so that the dataset could be directly loaded into CloneCompass. The steps for data preparation were as follows.

1. **Clone detection:** I used CCFinder to compare the files in the Linux kernel v5.0.9, generating a *.ccfxd* file. A *.ccfxd* file is a binary file created by CCFinder that contains identified clone pairs.

2. **Similarity generation:** I used a script[3] created by German et al. [55] to calculate the similarity between two files based on a *.ccfxd* file. The similarity

[1]The tool was downloaded from an open-source project: https://github.com/dmgerman/ccfx/tree/dmg.

[2]This version is downloaded from http://cdn.kernel.org/pub/linux/kernel/v5.x/.

[3]This is a open source project: https://github.com/dmgerman/plGrading/tree/master/ccfinderSibling

of file A (we refer to this as "source file") to B (we refer to this as "target file") is defined as the proportion of tokens in A found in B. As a result, 6,274 similarities (i.e., file pairs) were identified from 1,984 files.

3. **Pre-processing:** As mentioned above, I used four attributes to aggregate the TreeMap Matrix: 1) the file size, 2) the number of contributors, 3) the year created, and 4) the number of commits. However, CCFinder only provides one attribute, file size (i.e., the token number of each file). The remaining attributes were extracted from the Linux kernel using another tool: cregit [56]. To allow CloneCompass to deal with the four attributes, we revised our previous pre-processing steps designed for Kam1n0 and then generated an SQLite dataset that CloneCompass can load.

8.2 Linux Case Study Findings

After I obtained the SQLite dataset, I imported the dataset to CloneCompass to show the clones in the Linux kernel. As mentioned, the files in the dataset have four attributes: the file size (i.e., the number of tokens identified by CCFinder), the number of contributors, the year of each file created, and the number of commits. I explored the TreeMap Matrix and Adjacency Matrix with files ordered and aggregated according to the four attributes. Since CloneCompass allows us to set a similarity threshold (i.e., only similarities larger than the threshold could be displayed in the visualizations), in this case study, I set the similarity threshold to be 50% as an example. The findings based on my exploration are discussed below.

8.2.1 Aggregated by File Size

By inspecting the TreeMap Matrix aggregated by file size (see Fig. 8.1), I obtained the following insights:

1. **More clones were found in files with a similar size**: By observing the clones in area 1 (Diagonal) of Fig. 8.1, I found that clones are more likely to be found in files with a similar size, because many large and dark rectangles are shown along the diagonal. This result implies that some cloning activities might have occurred, e.g., copying a file with a small modification or addition. For example, to implement the same functionality but for different architectures, a

developer may clone a file that already worked for the functionality and then modify or add a few lines of code to adapt to the other architecture.

2. **Small files contain most of the clones**: Two TreeMaps in area 2 of Fig. 8.1 contain 414 and 177 similar file pairs, respectively, which implies that most clones are found in small files. A possible explanation might be that small files are usually easy to understand as these files are often implemented for simple functionalities, so it is likely that small files can be easily copied.

3. **Clones between large files and small files – small files as targets**: In area 3 of Fig. 8.1, some fragments in large files are similar to the small files, because the token number of target files is in the range of 52 to 239 (i.e., the top-left node on the y-axis), and the token number of source files is in the range of 1942 to 28,551 (i.e., the top-right node on the x-axis). This result indicates that a small file may be copied to a large file, or some fragments of a large file may be copied to a small file. Developers may have a particular reason for this activity.

4. **Clones between large files and small files – large files as targets**: The greyed out rectangles in area 4 of Fig. 8.1 show that similarities of small files to large files are not easy to find. This result may be due to the similarity calculation in this case study—a small number (the token number of a small file is small) divided by a very large number (the token number of a large file is large), generating a very small similarity (e.g., 1%). Since I set the similarity threshold to 50%, the small similarity scores are filtered out and cannot be seen in the result.

5. **Low similarity between large files**: Two TreeMaps in areas 5 and 6 of Fig. 8.1 only include seven file pairs in total, indicating that similar large files are not frequently found. Large files are usually more complex than small files, so copying a large file relies on a good understanding of the file and the system, which may be time consuming and risky for a developer. This might be the reason why large files are perhaps not copied.

I also considered the detailed information of a single TreeMap in this case study. For example, I looked at the TreeMap in area 3 of Fig. 8.1, selected this TreeMap and switched to the Adjacency Matrix, which is shown in Fig. 8.2. In this figure,

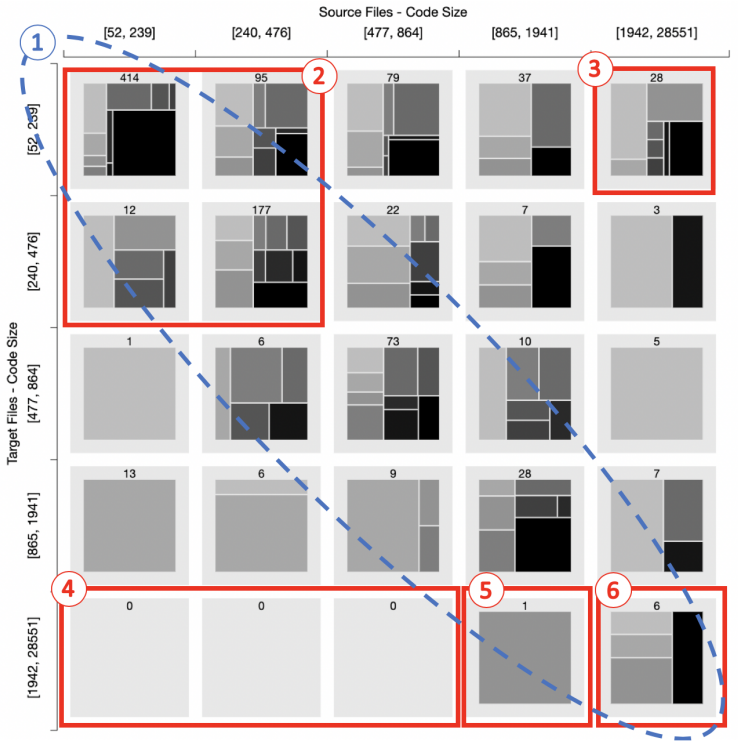


Figure 8.1: TreeMap Matrix aggregated by file size

small and large files are shown along the rows and columns, respectively. Three clone pairs with very high similarities (shown in the red rectangles) are found by comparing three source files named *ptrace.c* to a target file also named *ptrace.c*. By checking the full path of these files, we found that different *ptrace.c* files with similar code are used across different architectures. *Ptrace* includes interfaces used by debuggers and tracing tools[4]. To have a deeper look at the *Ptrace* clones, I searched for files named *ptrace.c* in Linux's source code, and found a *ptrace.c*[5] file which contains "common interfaces for ptrace()". In this *ptrace.c* file, a comment indicates that this file was created because "we do not want to continually duplicate across every architecture", implying that there may be a reason why those similar *ptrace.c* files across different architectures are not abstracted to the *ptrace.c* file with common interfaces.

I also considered the clones in areas 5 and 6 in Fig. 8.1. The two TreeMaps in the two areas present large clones, as the token number of target files is in the range of 865 to 28,551, and the token number of source files is in the range of 1942 to 28,551. First, I looked into area 5 and switched to the Adjacency Matrix, and found two files

[4]https://en.wikipedia.org/wiki/Ptrace
[5]https://github.com/torvalds/linux/blob/master/kernel/ptrace.c

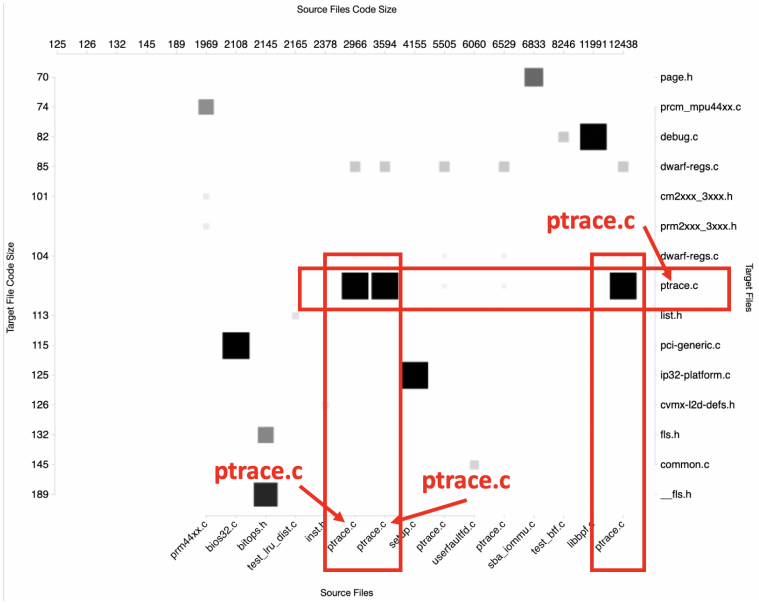


Figure 8.2: Adjacency Matrix ordered by the number of tokens in Linux files

with a similarity of 60%: *ptrace-syscall.c*[6] written for PowerPC and *ptrace_syscall.c*[7] written for x86. After I checked the source code, I found a comment in *ptrace-syscall.c* indicating the code in *ptrace-syscall.c* is "heavily based on *ptrace_syscall.c* and was adapted to run on PowerPC".

I further considered area 6 and switched to the Adjacency Matrix, and found a large clone pair in two files, both of which were named insn.c[8]. I did not find the sources of both files in the Linux GitHub repository because the Linux developers have taken care of this clone in the latest version (a newer version compared to v5.0.9 that I used for this case study). According to the commit history[9], the two insn.c files are identical and only one of them is needed under the tools subdirectory, so the Linux developers moved this file to a shared location and eliminated the clone.

8.2.2 Aggregated by Number of Contributors

The TreeMap Matrix aggregated by the number of contributors is shown in Fig. 8.3. From the TreeMaps, we could easily see:

[6] /tools/testing/selftests/powerpc/ptrace/ptrace-syscall.c
[7] /tools/testing/selftests/x86/ptrace_syscall.c
[8] /tools/perf/util/intel-pt-decoder/insn.c and /tools/objtool/arch/x86/lib/insn.c
[9] https://github.com/torvalds/linux/commit/d046b725487a97a3a3b35a00e84ca093963b8b4e#diff-21a9984a5e4727f6ce3df1a418f3ffa1

1. Files contributed by only one author contain a high number of file pairs—106 (see the top-left TreeMap).

2. Files with a similar number of contributors tend to have more clones.

3. Files contributed by more than eight people contain a relatively small number of clones, and most of the clones are found in the files with contributors fewer or equal to eight.

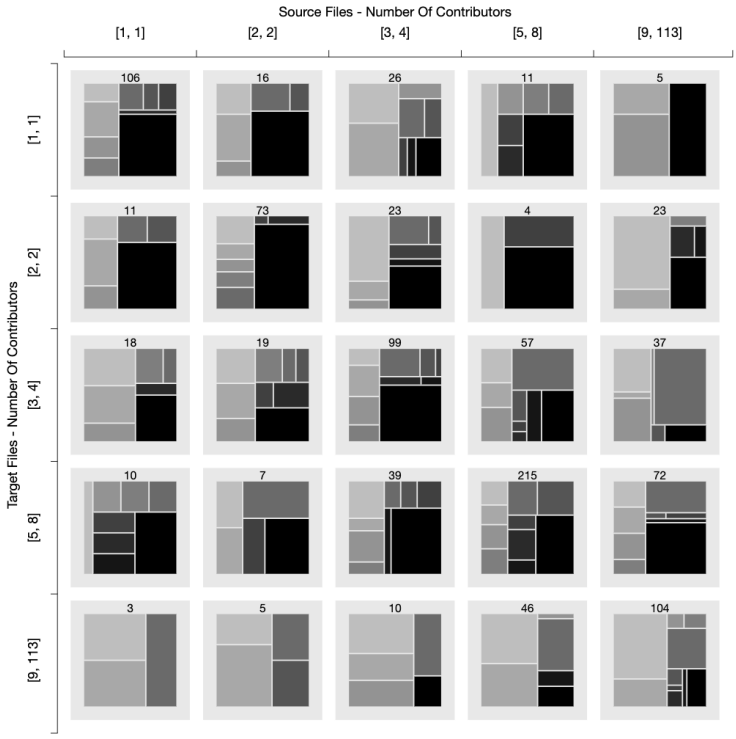


Figure 8.3: TreeMap Matrix aggregated by the number of contributors

A possible reason for the findings may be that files modified by many contributors may have many small code snippets added by each of the contributors, so these files are less likely to be similar to other files. In contrast, the files modified by a small number of people may have more chances to be similar to other files.

Furthermore, similarities between files with a large number of contributors and files with a small number of contributors drew our attention, so I investigated the top-right TreeMap Matrix and switched to Adjacency Matrix. I found that a file named *pci-generic.c* (written by one author) and a file named *bios32.c* (written by 24 authors) have a similarity of 99%. After checking the source code, I found that

the contributor of *pci-generic.c*[10] used the code in *bios32.c*[11], and this contributor commented that a function was "taken from *pci-generic.c*", but the reason for this cloning activity was not provided.

8.2.3 Aggregated by Number of Commits

The TreeMap Matrix aggregated by the number of commits is shown in Fig. 8.4. After my exploration, I found that most clones exist in files with a number of commits less than 27, and files with high commits are less likely to have clones. The reason might be the same as the one we obtained from the TreeMap Matrix aggregated by the number of contributors.

To determine the relationship between the number of commits and the number of contributors, I calculated the Spearman and Pearson correlation coefficient [57] of the two attributes. As a result, the Spearman correlation coefficient is 0.919 and the Pearson correlation coefficient is 0.866, indicating that the two attributes have a monotonic relationship. This result further proves the hypothesis that files contributed by many people can generate more commits, and such files are less likely to contain clones.

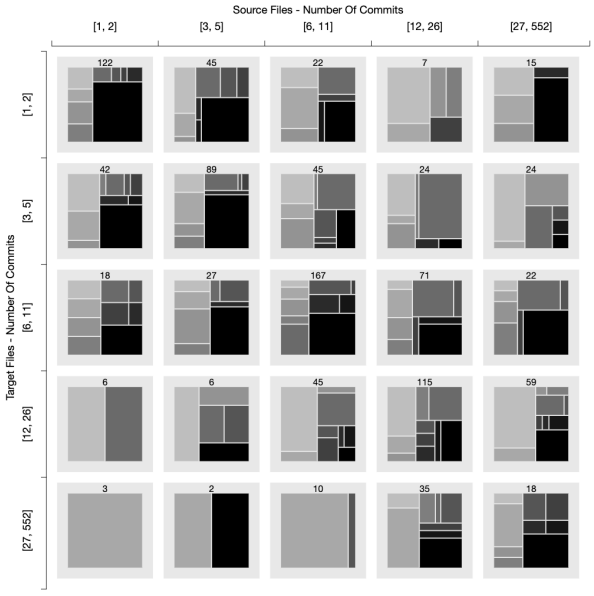


Figure 8.4: TreeMap Matrix aggregated by number of commits

[10]https://github.com/torvalds/linux/blob/master/arch/mips/pci/pci-generic.c
[11]https://github.com/torvalds/linux/blob/master/arch/arm/kernel/bios32.c

8.2.4 Aggregated by Year Created

The TreeMap Matrix aggregated by the year created contains two pages of TreeMaps. Fig. 8.5 presents the result of the first page. By exploring these TreeMaps, I found that files created in the same or similar years tend to have more clones, indicating that developers may prefer to copy code added in recent years.

Additionally, I looked into clones in very different years. For example, two files were created in 2003 and 2016. With a closer look at the comparison between a file *signal_n32.c*[12] created in 2003 and a file *signal_o32.c*[13] created in 2016, we realized that the two files include similar code written for different CPUs.

To summarize, Chapter 7 and 8 describe a preliminary evaluation I conducted with our stakeholders and a Linux case study to evaluate CloneCompass. The results showed that CloneCompass can help users explore code clones and obtain some insights to assist with their tasks. However, from the results, I also identified a number of limitations, which are discussed in the next chapter.

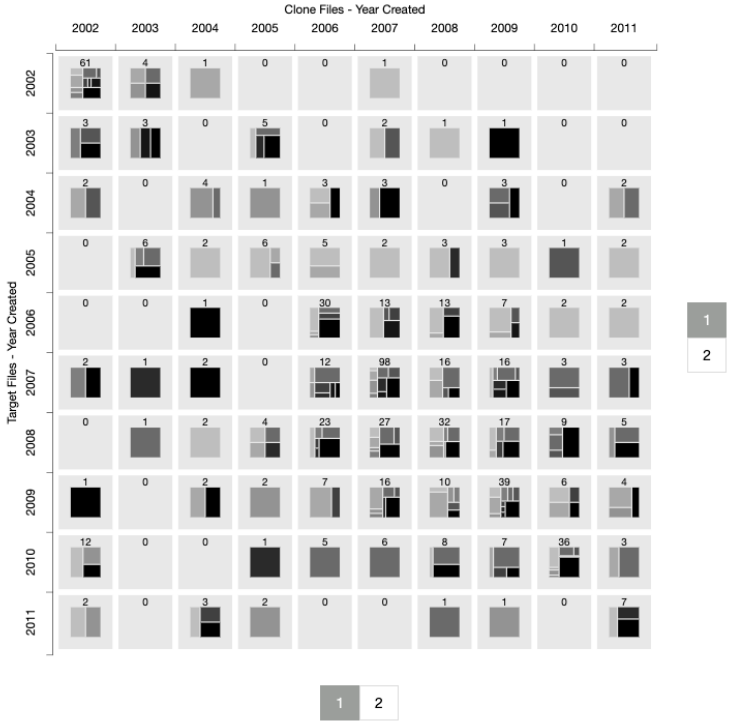


Figure 8.5: TreeMap Matrix aggregated by year created

[12]https://github.com/torvalds/linux/blob/master/arch/mips/kernel/signal_n32.c
[13]https://github.com/torvalds/linux/blob/master/arch/mips/kernel/signal_o32.c

Chapter 9

Discussion and Future Work

According to the results of the preliminary evaluation and the Linux case study, I identified the applicability and advantages of the tool, CloneCompass. From the results, I also found a number of limitations and I point out possible future work.

9.1 Discussion of Preliminary Evaluation and Linux Case Study Results

Based on the results of the preliminary evaluation, the combination of a TreeMap Matrix and an Adjacency Matrix in CloneCompass enables users to explore any clones in a dataset, even if the dataset contains over 90 million function-clone pairs. By using CloneCompass, users can gain a variety of insights, which include but are not limited to: 1) finding clones based on code size or block size; 2) identifying clusters of high, low, or relatively different similarities; 3) discovering new instances of cloning; 4) identifying clones that are static over time periods; and 5) comparing clones of multiple binaries or versions. Such insights can help users understand how clones are dispersed across many systems.

In addition, CloneCompass was used to show similar files in Linux, demonstrating that the tool may be applicable to clones detected by CCFinder [37]. During the exploration of the Linux clones, I obtained a number of insights as discussed in the previous chapter. This case study also shows that CloneCompass is potentially capable of analyzing clones detected by other techniques other than Kam1n0 and CCFinder.

Furthermore, compared with other matrix-based views [25][26][45][46][47], the

TreeMap Matrix originally created in this study has three advantages. First, in contrast with one-dimensional visual representations (e.g., color shade), the two-dimensional TreeMap view contains rich information, such as the number of clone pairs with different similarity ranges. Second, compared with other two-dimensional visual data representations (e.g., histograms), a TreeMap saves space on a size-limited display: a TreeMap uses almost all the space in a cell and is highly readable. Third, the use of across-TreeMap multiple selections in the TreeMap Matrix allows users to select data from any area in the TreeMap Matrix and then inspect selected data in a detailed view. In contrast, the zooming technique used in existing tools only allows the user to see the data from a limited area in a detailed view. The use of multiple selections addresses the space limitation of zooming and allows the user to choose and zoom in on any area.

However, CloneCompass also has a number of limitations, which are discussed next.

9.2 Limitations and Future Work

Evaluation

The main limitation of this study is that the two participants of the preliminary evaluation are Kam1n0's creators, not security analysts or reverse engineers. Even though Kam1n0's creators know their work quite well, a bias could exist in their evaluation as they may be inclined to be positive about CloneCompass. However, I encouraged them throughout the process to give us honest feedback about the proposed solution (which they did with earlier iterations of our design). Unfortunately, because security analysts are busy and often work on secret projects, I could not find other participants for the study (although they indicated wanting to use CloneCompass to support their work). Therefore, in the future, we hope to find security analysts or reverse engineers to validate the usability of CloneCompass in security analysis.

In the Linux case study, although I obtained many insights by using CloneCompass to explore the clones in Linux, the reason for the results are not clear, because I am not a Linux expert. Therefore, it would be worthwhile to conduct an evaluation with Linux experts to validate CloneCompass, which could help us further validate the tool.

Tool Improvements

Based on the results of the preliminary evaluation and the Linux case study, I identified possible improvements to the visualizations in CloneCompass:

- **Content of an entity:** CloneCompass does not include the content of an entity, such as the code in a function or a file. However, it is likely that the user may need to see the code after they find two similar functions or files, which could be helpful for further investigations. In the future, we may implement functionality that allows the user to see the content of entities, e.g., connecting CloneCompass to Kam1n0.

- **Short-term loss of context in the switches between two matrix-based views:** Users can forget what they have visited after several switches between the TreeMap Matrix and the Adjacency Matrix. This problem can be even more serious for large-scale datasets because a large number of TreeMaps could exist in a TreeMap Matrix. However, CloneCompass does not provide any information about the visited data. Therefore, an annotation technique or a browsing history will be needed to help the user remember what they have visited.

- **Pagination:** When CloneCompass contains a large number of pages, it is not easy to find a target combination of two pages. To solve this problem, a tooltip with guiding information can be added when hovering over a page. Such information may be helpful to select a page.

- **Scalability limits:** CloneCompass has scalability limits in both the TreeMap Matrix and Adjacency Matrix. There is a maximum number of TreeMaps that a TreeMap Matrix can load in CloneCompass, and the maximum is unknown because of the limited access to larger datasets. As a result, it is possible that a TreeMap Matrix aggregated by binaries fails to load if the number of binaries is too large. Also, since an Adjacency Matrix shows a subset of clone pairs from a TreeMap Matrix, when the selected rectangles in the TreeMap Matrix contains a large number of clone pairs, the Adjacency Matrix could take a long time to display. In the future, scalability limits of the TreeMap Matrix and Adjacency Matrix can be improved by optimizing the code performance of CloneCompass.

- **Generalization:** When I applied CloneCompass to show the clones in Linux, several additional steps were performed to process the data, implying that the proposed tool is not general enough and cannot be directly used to show clones in any dataset. For example, in order to show the TreeMap Matrix aggregated by the number of contributors, which were not included in the original implementation (i.e., CloneCompass designed for Kam1n0), I had to modify a small portion of code. Therefore, generalizing CloneCompass to other applications requires future work.

Chapter 10

Conclusions

In this thesis, I reported on a design study to support code clone analysis. I characterized the challenges that users experienced when they explored large-scale search results from Kam1n0 (an assembly code clone search engine). I then iteratively designed and refined a novel interactive visual tool, CloneCompass, and later conducted a preliminary evaluation as part of a design study. I also carried out a Linux case study to extend the applicability of CloneCompass.

CloneCompass' design pairs a TreeMap Matrix view with an Adjacency Matrix view to show and support the exploration and inspection of a large number of code clones. The results from a preliminary evaluation show that users can gain useful insights while using CloneCompass during their exploration of assembly code clones. The Linux case study demonstrates the potential of CloneCompass to be used for other duplicated information.

This study makes the following contributions:

1. **CloneCompass:** I created CloneCompass which pairs a novel TreeMap Matrix view and an Adjacency Matrix view for clone analysis.

2. **Problem characterization:** I characterized problems encouraged by a group of stakeholders, which may also be experienced by other security analysts and reverse engineers.

3. **Preliminary evaluation:** I conducted a preliminary evaluation to validate the usability of CloneCompass.

4. **Linux case study:** I conducted a Linux case study to show the applicability of CloneCompass to other domains.

In future research, we intend to concentrate on optimizing CloneCompass using our evaluation insights. More work will be needed to verify CloneCompass' usability in reverse engineering. Further research into the application of the novel TreeMap Matrix paired with an Adjacency Matrix for non-assembly clone analysis is also a plausible next step.

Appendix A

Binning Process

In a TreeMap Matrix, functions can be aggregated by one of the four attributes extracted from Kam1n0: code size, block size, binary, and version. If functions are aggregated by code size or block size, the functions are first ordered by the attribute and then divided into distinct ranges by a binning algorithm described as follows.

1. Receive N functions from one or multiple binaries.
2. Order the functions by a quantitative attribute, i.e., code size or block size.
3. Divide N functions into A bins, and each bin contains M functions, where $A = ceiling(N/M)$. This step generates a list of bins, where each bin contains M functions.
4. Mark the last function in the first bin as $STOP$, and the corresponding index is $STOP\_INDEX$.
5. If the code size of function $STOP$ is equal to the code size of the function at the position of $STOP\_INDEX + 1$, find the closest function with a code size that is unequal to code size of function $STOP$ from the tail of the first bin to the beginning of the next bin. The purpose of this step is to ensure that functions with a same code size can be grouped into the same bin.
6. Update $STOP$ as the function at step 5 and the corresponding index $STOP\_INDEX$. Find the functions from the beginning to $STOP\_INDEX$ and allocate these functions into a bin.

7. Remove the function found at step 6 from the function list. Start at the next function at $STOP\_INDEX + 1$ and repeat steps 3-7 until all functions are allocated into bins.

Appendix B

Adjacency Matrix Using Different Color Schemes

As described in Chapter 6, CloneCompass supports the choice of three color schemes in an Adjacency Matrix. Examples of the Adjacency Matrix using different color schemes are shown as follows.

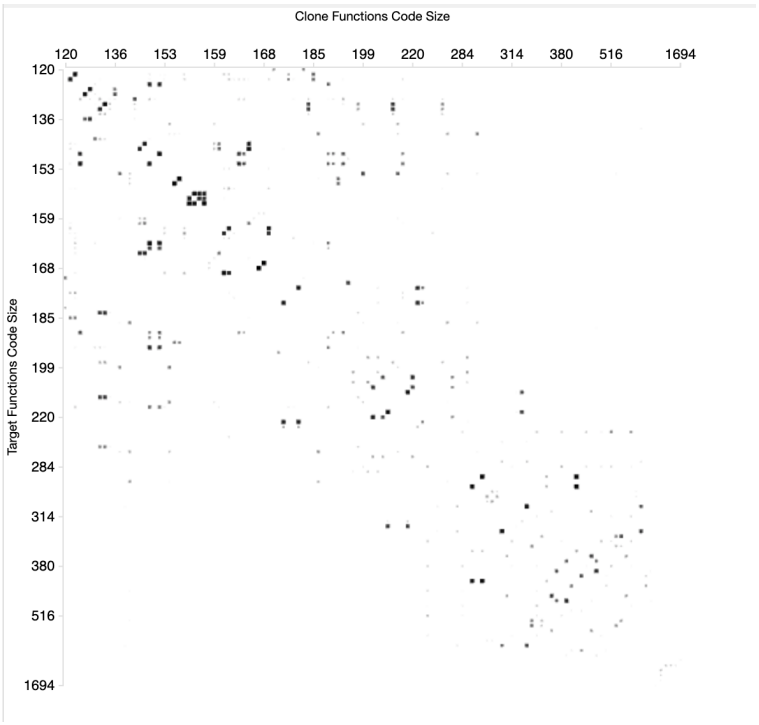


Figure B.1: Adjacency Matrix using BrBG color scheme

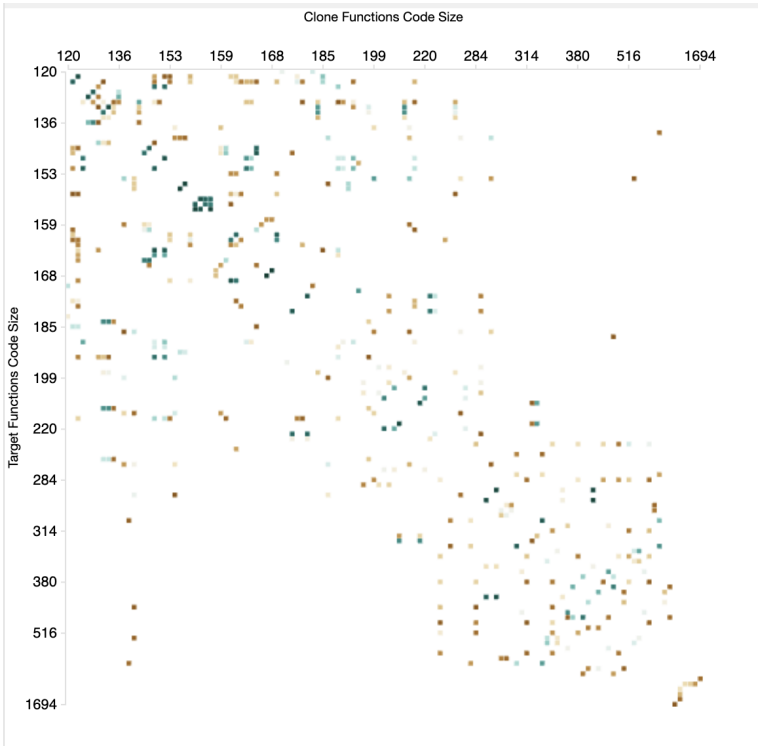


Figure B.2: Adjacency Matrix using BrBG color scheme

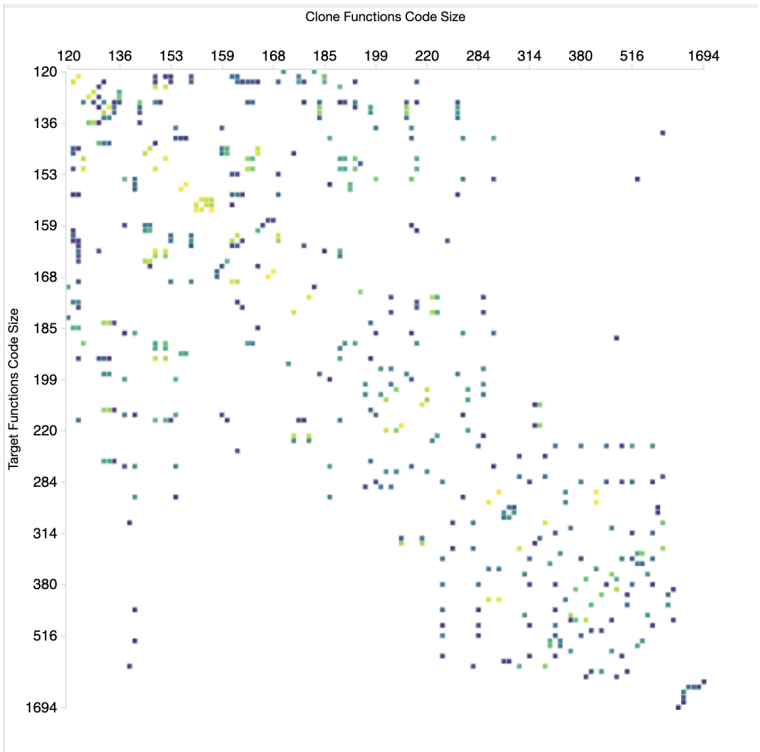


Figure B.3: Adjacency Matrix using Viridis color scheme

Appendix C

Questions

The following questions were asked after participants used CloneCompass in the preliminary evaluation described in Chapter 7.

1. What useful information can you get from Treemap Matrix?
2. What useful information can you get from Adjacency Matrix?
3. Is there any information you cannot find in Treemap Matrix but you were expected to see?
4. Is there any information you cannot find in Adjacency Matrix but you were expected to see?
5. Is the tool useful in your code clone analysis?
6. Has the tool helped you solve all your problems? What problems have or have not been solved?
7. Is the Treemap Matrix easy to read and use?
8. Is the Adjacency Matrix easy to read and use?
9. Is the tool user-friendly?
10. What features would you like to add in the future?
11. Do you have any suggestions and comments?

Appendix D

Kam1n0 Search Result Example

The result of a search result from Kam1n0 is saved in a JSON file. An example of JSON file is shown below.


```
{
    "function": {
        "binaryName": "libz.so.1.2.11-gcc-g-O0-m32-fno-pic.bin",
        "binaryId": "-2069062410952586386",
        "functionName": ".init_proc",
        "functionId": "7574097512985699929",
        "startAddress": "5992",
        "blockSize": 3,
        "codeSize": 11
    },
    "clones": [
        {
            "functionId": "-8768255973834564267",
            "functionName": ".init_proc",
            "binaryId": "8154864172249410757",
            "binaryName": "libz.so.1.2.11-gcc-g-O2-m32-fno-pic.bin",
            "numBbs": 3,
            "codeSize": 11,
            "similarity": 0.9519564010326861,
            "clonedParts": []
        },
        {
            "functionId": "3864173850524068438",
            "functionName": ".init_proc",
            "binaryId": "6272564911413261260",
            "binaryName": "libz.so.1.2.11-gcc-g-O1-m32-fno-pic.bin",
            "numBbs": 3,
            "codeSize": 11,
            "similarity": 0.9511705898516535,
            "clonedParts": []
        }
    ]
}
```


Figure D.1: An example of a search result generated by Kam1n0

Bibliography

[1] N. Gehlenborg and B. Wong, "Points of view: networks," 2012.

[2] S. Livieri, Y. Higo, M. Matushita, and K. Inoue, "Very-large scale code clone analysis and visualization of open source programs using distributed ccfinder: D-ccfinder," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 106–115.

[3] G. Zhang, X. Peng, Z. Xing, and W. Zhao, "Cloning practices: Why developers clone and what can be changed," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 285–294.

[4] D. Chatterji, J. C. Carver, and N. A. Kraft, "Cloning: The need to understand developer intent," in *Proceedings of the 7th International Workshop on Software Clones*. IEEE Press, 2013, pp. 14–15.

[5] C. J. Kapser and M. W. Godfrey, "cloning considered harmful considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, p. 645, 2008.

[6] C. K. Roy, M. F. Zibran, and R. Koschke, "The vision of software clone management: Past, present, and future (keynote paper)," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE, 2014, pp. 18–33.

[7] M. Sedlmair, M. Meyer, and T. Munzner, "Design study methodology: Reflections from the trenches and the stacks," *IEEE transactions on visualization and computer graphics*, vol. 18, no. 12, pp. 2431–2440, 2012.

[8] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[9] M. Asaduzzaman, "Visualization and analysis of software clones," Ph.D. dissertation, University of Saskatchewan, 2012.

[10] G. Casazza, G. Antoniol, U. Villano, E. Merlo, and M. Di Penta, "Identifying clones in the linux kernel," in *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation.* IEEE, 2001, pp. 90–97.

[11] M. W. Godfrey and D. M. German, "The past, present, and future of software evolution," in *2008 Frontiers of Software Maintenance.* IEEE, 2008, pp. 129–138.

[12] A. Gupta and B. Suri, "A survey on code clone, its behavior and applications," in *Networking Communication and Data Knowledge Engineering.* Springer, 2018, pp. 27–39.

[13] M. 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.* IEEE, 2004, pp. 83–92.

[14] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey, "Cloning by accident: an empirical study of source code cloning across software systems," in *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE, 2005, pp. 10–pp.

[15] N. Saini, S. Singh *et al.*, "Code clones: Detection and management," *Procedia computer science*, vol. 132, pp. 718–727, 2018.

[16] H. Min and Z. Li Ping, "Survey on software clone detection research," in *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences.* ACM, 2019, pp. 9–16.

[17] D. Chatterji, J. C. Carver, and N. A. Kraft, "Code clones and developer behavior: results of two surveys of the clone research community," *Empirical Software Engineering*, vol. 21, no. 4, pp. 1476–1508, 2016.

[18] S. H. Ding, B. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM, 2016, pp. 461–470.

[19] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization*. IEEE, 2019, p. 0.

[20] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1798–1828, 2013.

[21] H. A. Basit, M. Hammad, and R. Koschke, "A survey on goal-oriented visualization of clone data," in *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*. IEEE, 2015, pp. 46–55.

[22] M. Ghoniem, J.-D. Fekete, and P. Castagliola, "A comparison of the readability of graphs using node-link and matrix-based representations," in *IEEE Symposium on Information Visualization*. Ieee, 2004, pp. 17–24.

[23] ——, "On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis," *Information Visualization*, vol. 4, no. 2, pp. 114–135, 2005.

[24] R. Keller, C. M. Eckert, and P. J. Clarkson, "Matrices or node-link diagrams: which visual representation is better for visualising connectivity models?" *Information Visualization*, vol. 5, no. 1, pp. 62–76, 2006.

[25] F. Beck and S. Diehl, "Visual comparison of software architectures," *Information Visualization*, vol. 12, no. 2, pp. 178–199, 2013.

[26] J. Abello and F. Van Ham, "Matrix zoom: A visual interface to semi-external graphs," in *IEEE symposium on information visualization*. IEEE, 2004, pp. 183–190.

[27] E. Adar and M. Kim, "Softguess: Visualization and exploration of code clones in context," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 762–766.

[28] A. Telea and D. Auber, "Code flows: Visualizing structural evolution of source code," in *Computer Graphics Forum*, vol. 27, no. 3. Wiley Online Library, 2008, pp. 831–838.

[29] F. Chevalier, D. Auber, and A. Telea, "Structural analysis and visualization of c++ code evolution using syntax trees," in *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*. ACM, 2007, pp. 90–97.

[30] B. Hauptmann, V. Bauer, and M. Junker, "Using edge bundle views for clone visualization," in *Proceedings of the 6th International Workshop on Software Clones*. IEEE Press, 2012, pp. 86–87.

[31] A. Hanjalić, "Clonevol: Visualizing software evolution with code clones," in *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2013, pp. 1–4.

[32] M. Asaduzzaman, C. K. Roy, and K. A. Schneider, "Viscad: flexible code clone analysis support for nicad," in *Proceedings of the 5th International Workshop on Software Clones*. ACM, 2011, pp. 77–78.

[33] K. Yoshimura and R. Mibe, "Visualizing code clone outbreak: An industrial case study," in *2012 6th International Workshop on Software Clones (IWSC)*. IEEE, 2012, pp. 96–97.

[34] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 117–128.

[35] Z. M. Jiang, A. E. Hassan, and R. C. Holt, "Visualizing clone cohesion and coupling," in *2006 13th Asia Pacific Software Engineering Conference (APSEC'06)*. IEEE, 2006, pp. 467–476.

[36] T. Munzner, *Visualization analysis and design*. AK Peters/CRC Press, 2014.

[37] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[38] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue, "Gemini: Maintenance support environment based on code clone analysis," in *Proceedings Eighth IEEE Symposium on Software Metrics*. IEEE, 2002, pp. 67–76.

[39] M. Sano, E. Choi, N. Yoshida, Y. Yamanaka, and K. Inoue, "Supporting clone analysis with tag cloud visualization," in *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*. ACM, 2014, pp. 94–99.

[40] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue, "Method and implementation for investigating code clones in a software system," *Information and Software Technology*, vol. 49, no. 9-10, pp. 985–998, 2007.

[41] S. Livieri, Y. Higo, M. Matsushita, and K. Inoue, "Analysis of the linux kernel evolution using code clone coverage," in *Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007)*. IEEE, 2007, pp. 22–22.

[42] J. R. Cordy, "Exploring large-scale system similarity using incremental clone detection and live scatterplots," in *2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 2011, pp. 151–160.

[43] B. Shneiderman, "The eyes have it: A task by data type taxonomy for information visualizations," in *The craft of information visualization*. Elsevier, 2003, pp. 364–371.

[44] L. Wang, G. Wang, and C. A. Alexander, "Big data and visualization: methods, challenges and technology progress," *Digital Technologies*, vol. 1, no. 1, pp. 33–38, 2015.

[45] F. Van Ham, "Using multilevel call matrices in large software projects," in *IEEE Symposium on Information Visualization 2003 (IEEE Cat. No. 03TH8714)*. IEEE, 2003, pp. 227–232.

[46] F. Van Ham, H.-J. Schulz, and J. M. Dimicco, "Honeycomb: Visual analysis of large scale social networks," in *IFIP Conference on Human-Computer Interaction*. Springer, 2009, pp. 429–442.

[47] N. Elmqvist, T.-N. Do, H. Goodell, N. Henry, and J.-D. Fekete, "Zame: Interactive large-scale graph visualization," in *2008 IEEE Pacific visualization symposium*. IEEE, 2008, pp. 215–222.

[48] R. H. Von Alan, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[49] A. H. JØRGENSEN, "Thinking-aloud in user interface design: a method promoting cognitive ergonomics," *Ergonomics*, vol. 33, no. 4, pp. 501–507, 1990.

[50] T. Von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. J. van Wijk, J.-D. Fekete, and D. W. Fellner, "Visual analysis of large graphs: state-of-the-art and future research challenges," in *Computer graphics forum*, vol. 30, no. 6. Wiley Online Library, 2011, pp. 1719–1749.

[51] Z. Liu, S. B. Navathe, and J. T. Stasko, "Network-based visual analysis of tabular data," in *2011 IEEE Conference on Visual Analytics Science and Technology (VAST)*. IEEE, 2011, pp. 41–50.

[52] T. Jankun-Kelly, T. Dwyer, D. Holten, C. Hurter, M. Nöllenburg, C. Weaver, and K. Xu, "Scalability considerations for multivariate graph visualization," in *Multivariate Network Visualization*. Springer, 2014, pp. 207–235.

[53] M. Nacenta, U. Hinrichs, and S. Carpendale, "Fatfonts: combining the symbolic and visual aspects of numbers," in *Proceedings of the international working conference on advanced visual interfaces*. ACM, 2012, pp. 407–414.

[54] C. Chang, B. Bach, T. Dwyer, and K. Marriott, "Evaluating perceptually complementary views for network exploration tasks," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 2017, pp. 1397–1407.

[55] D. M. German, M. Di Penta, Y.-G. Gueheneuc, and G. Antoniol, "Code siblings: Technical and legal implications of copying code between applications," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 81–90.

[56] D. M. German, B. Adams, and K. Stewart, "cregit: Token-level blame information in git version control repositories," *Empirical Software Engineering*, pp. 1–39, 2019.

[57] J. Hauke and T. Kossowski, "Comparison of values of pearson's and spearman's correlation coefficients on the same sets of data," *Quaestiones geographicae*, vol. 30, no. 2, pp. 87–93, 2011.