CloneCompass: Visualizations for Exploring Assembly Code Clone Ecosystems

Ying Wang, Jorin Weatherston, Margaret-Anne Storey, Daniel M. German
Department of Computer Science
University of Victoria
Victoria, Canada
yingwang2@uvic.ca, jorinw@uvic.ca, mstorey@uvic.ca, dmg@uvic.ca

***Abstract*—Assembly code analysis is an intensive process undertaken by security analysts and reverse engineers to discover vulnerabilities in existing software when source code is unavailable. 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 paper 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 show how CloneCompass enables these users to visually and interactively explore code clone data generated from software systems with suspected vulnerabilities.**

***Index Terms*—visualization, assembly code clone, Kam1n0, matrix-based view, reverse engineering**

I. Introduction

In software engineering, the security analysis of binaries requires the inspection of large systems without their source code. Instead, assembly is examined to identify security vulnerabilities. Analysts typically have to inspect many different applications, and each of them can consist of millions of assembly level instructions. Code clone detection is frequently used to improve the productivity of assembly code analysts: if a vulnerability is found in a code fragment, its clones are likely vulnerable as well. Similarly, if a code fragment is safe, its clones might be inspected for vulnerabilities with a lower priority.

Kam1n0 is an efficient assembly code clone search engine to support the analysis of vulnerabilities in binaries [1]. It is capable of finding cloned functions in a single binary and across a set of binaries. Kam1n0 uses a few different visualizations to support clone inspection:

1) 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 analysts search for a given function.

2) A detailed view shows the similarities between a target function and its clones.

3) 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 millions clone pairs. Using the current detailed and summary views 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 paper, we describe a visualization environment for assembly clone analysis called CloneCompass. CloneCompass implements a pairing of two visualizations for very large clone sets: a novel visualization TreeMap Matrix (see Fig. 1 (a)) and an Adjacency Matrix (see Fig. 1 (b)). CloneCompass provides many-to-many comparisons for large-scale datasets and enables reverse engineers[2] to efficiently explore clones identified by Kam1n0. Throughout the exploration, reverse engineers can gain insights about the software ecosystem under analysis and more efficiently find vulnerabilities, which is critical in security analysis. To identify Kam1n0 users' needs and find an efficient visual solution to address their issues, we conducted a design study [2] with Kam1n0's creators, researchers in the Data Mining and Security (DMaS) Lab at McGill University[3], and Kam1n0's end users, reverse engineers from Defence Research and Development Canada (DRDC). Our contributions include the design of a novel matrix-based TreeMap view which coordinates with an Adjacency Matrix for clone analysis (implemented in CloneCompass), a problem characterization for assembly clone analysis, and an initial evaluation of CloneCompass through which we deepened our understanding of the requirements for assembly clone analysis. A demo

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Defence Research and Development Canada (DRDC).

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

[2]We use the terms reverse engineer and security analyst interchangeably throughout the paper.

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

video of CloneCompass can be viewed at the following link https://youtu.be/IIcefibBalI.

II. Related Work

In this section, we focus on visualizations used in code clone analysis and matrix-based visualizations for large-scale graphs.

A. Visualization for Code Clone Analysis

Visualizations such as node-link diagrams and matrix-based views have been used for code clone analysis.

Many node-link diagrams are used during clone analysis according to different abstraction units. 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, module, versions, programs, etc. exist at the higher end of code abstraction.

SoftGUESS [3] uses node-link trees to show evolutionary code clone behaviors. The nodes in a node-link tree can be snippets, methods, classes, etc., and the edges can be dependencies, cloning relationships, etc. Although SoftGUESS is able to show over 8,000 nodes and 30,000 edges, the search results from Kam1n0 can contain millions of edges, so node-link trees do not scale to these needs. ClonEvol [4] uses radial trees to show changes across thousands of versions. However, ClonEvol only displays higher level abstraction units, such as projects, files, and scopes. A lower level abstraction unit (e.g., functions) can greatly increase the size of the graph that needs to be displayed because a file or project can contain a large number of functions. Icicle plot graphs have also been used to show lines of code, but only two versions of software can be compared with this approach [5] [6]. The results from Kam1n0 can contain multiple versions, so icicle plot graphs are not suitable for our needs.

Because of the need to present many-to-many comparisons, we chose to explore matrix-based views, which are commonly used in code clone analysis. D-CCFinder [7] uses scatter plots and color heatmaps to show frequently used code because they produce recognizable patterns. In another study, a heatmap was used to display the code clone coverage ratio between different versions of the Linux kernel [8]. A matrix-based view has also been used to compare software at multiple levels. For instance, live scatterplots [9] were implemented to analyze the similarities between two systems at multiple levels of data abstraction, such as files, directories, or subsystem directories. This prior work indicated that matrix-based views might be a good option for our purposes.

B. Matrix Views for Large-Scale Data

As the search results from Kam1n0 can contain millions of function-clone pairs, we consider here how matrix views may scale for large graphs.

The Adjacency Matrix is a typical matrix-based view used for many-to-many comparative analysis. Adjacency matrices are a practical way to display large, undirected networks [10], and they can scale up to 1,000 nodes and 1,000,000 edges [11]. However, the search results from Kam1n0 can consist of millions of edges, which is where the Adjacency Matrix falls short. Fortunately, many researchers have proposed some approaches for large matrix-based graphs, as discussed below.

There are limits of what humans can understand and what computers can display with current visualization techniques. One way to tackle these issues for large-scale data is to visualize data by showing an overview first and then providing details on demand [12] [13]. Several studies propose a zoomable Adjacency Matrix for hierarchical data, such as software architectures [14] [15] and phone traffic [16]. Hierarchical data is shown at multiple levels: an overview displays parent nodes (i.e., nodes at higher levels in the hierarchy) in rows and columns, and the comparison of two parent nodes is shown in the intersection of a row and a column in a zoomable Adjacency Matrix [16]. By zooming in on a specific area, a comparison between the child nodes (i.e., nodes at lower levels in the hierarchy) of specified parent nodes is shown on the screen.

For data without a hierarchical structure, aggregation techniques can be applied to abstract the nodes into groups, with each group containing multiple nodes. MatrixExplorer [17] is one technique that shows the distribution of aggregated nodes in an overview. The overview contains compact ordered rectangles, with the size and color of each rectangle indicating the number of aggregated nodes. In other approaches, the overview uses a matrix-based structure but the nodes at rows and columns are replaced by aggregated nodes. For instance, Honeycomb [18] introduces an aggregation technique to collapse multiple cells into one cell and uses color luminance to show the density of the edges in that area.

ZAME [19] provides eight different glyphs used 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, allowing the Adjacency Matrix to show millions of nodes and billions of edges.

III. Research Methodology

The goal of our research was to design an interactive visual tool that could efficiently show a large number of assembly code clones from Kam1n0 to help users better understand the extent of clones in a software ecosystem and find vulnerabilities. We used a design study methodology [2], an increasingly prominent approach used in problem-driven visualization. In a design study, researchers cooperate closely with domain experts and gain problem insights through ongoing discussions with stakeholders. These problem insights guide the researchers to refine and improve their design.

Over a period of eight months, we collaborated with Kam1n0's creators, researchers in the Data Mining and Security (DMaS) Lab at McGill University. 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

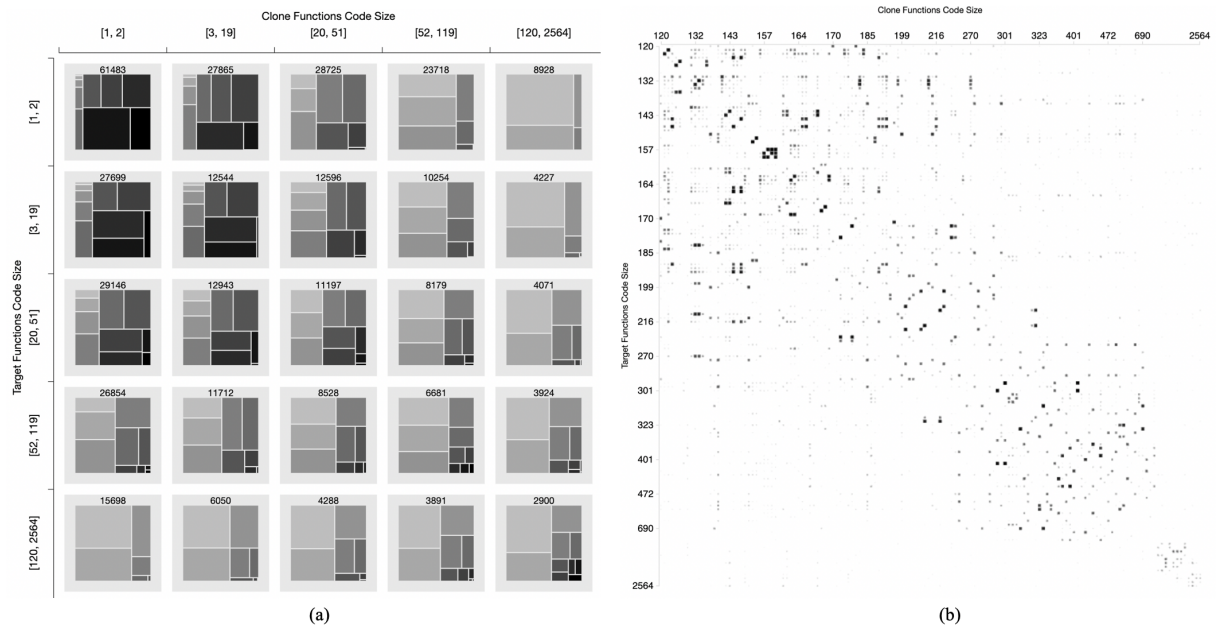

Fig. 1. Visualizations used in CloneCompass: (a) A Treemap Matrix overview. (b) Details in the Adjacency Matrix showing the data from selected rectangles in the Treemap Matrix in (a)

problem points. As we exposed their issues, we developed requirements for new visualizations that would scale to our setting, and iteratively designed, implemented and refined our visualization tool, CloneCompass, based on their feedback. During this work, we initially proposed an Adjacency Matrix to show the dataset, but it could not efficiently load the data because it did not scale to the millions of clones from Kam1n0's search results. The process of improving CloneCompass also helped us develop our understanding of the problem faced by our stakeholders, as we discuss below.

In the final stages of our study, we conducted a preliminary evaluation to validate CloneCompass. During this evaluation, we used a "think-aloud protocol" [20] to help us understand the participants' cognitive processes while they used CloneCompass, and to determine the tool's usability and ability to assist with the analysis tasks.

IV. Problem Characterization

Through our collaboration with Kam1n0's designers and end users, we identified key challenges that these people face when browsing search results from Kam1n0. Below, we describe the Kam1n0 tool and how it is used, and then describe the challenges that emerged from our research.

Kam1n0 is an efficient and scalable assembly code clone search engine [1]. It allows the user to build a *clone-search application*[4] and then index one or multiple binaries in this application to generate a repository. The user can then search for either a single assembly functions (we refer to these as a "target function") or multiple functions in a binary file (we refer to this as a "target binary"), and then compare these functions to the functions in the repository.

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

Kam1n0 allows the user to build multiple *clone-search applications* and run multiple searches in each application. Each search result contains the similarities between target functions and the functions in the repository. Before each search, the user can set a similarity threshold, where only similarities larger than the threshold are shown in a search result.

A search result for target functions and their clones in a *clone-search application* are presented in three different ways.

1) A flow graph view, a text diff view, and a clone group view[4] are used to compare the subgraphs of two functions. These views are useful for comparing the details of two functions when the user searches for a given function. In this paper, we focus on a higher level of abstraction (e.g., functions and binaries) and not the function subgraphs.

2) A detailed view shows target functions and clones, see Fig. 2. A list of target functions are shown in Fig. 2 (a), where 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. 2 (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 function ordered by similarity (according to the user-specified threshold as mentioned above). In Fig. 2 (c), the centre node represents the same target function in Fig. 2 (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 shown in Fig. 2 (a) can be very long. When a target function contains a large number of clones, the clone list in Fig. 2 (b) can also be very long. Exploring these long lists of target functions and/or clones can be very time

consuming.

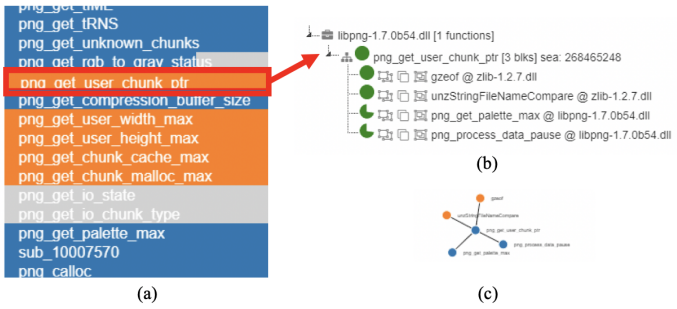

Fig. 2. Detail view in Kam1n0: (a) Target functions list. (b) A detail view shows a target function and its clones. (c) A node-edge graph view shows a target function and its clones.

3) A summary view (Fig. 3) shows statistics about the clones of a target binary and the binaries in the repository. For example, binary A has $x\%$ clones with a similarity score more than $y\%$ compared with binary B, where $y\%$ is the similarity threshold set by users. 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 these two binaries, they need to use the detail view shown in Fig. 2.

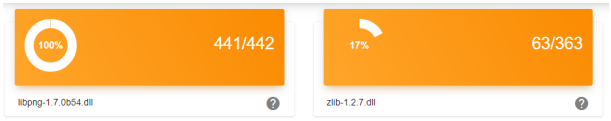

Fig. 3. Summary view in Kam1n0

Through interviews and ongoing discussions with our stakeholders, and during the refinement of our interim prototype visualizations, we discovered that the users experienced three key challenges when using Kam1n0's detail and summary views:

- **Many-to-many comparisons:** When using Kam1n0, the 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 one detail view only contains a single target function and one summary view only contains a single target binary. A suitable visualization can show such many-to-many comparisons, but scalability must be considered.
- **Scalability of clone exploration:** As mentioned above, it is difficult to efficiently explore clones because a search result from Kam1n0 can contain a very large number of assembly clones. The number can be 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, the reverse engineers are expected to have a comprehensive 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/biggest clone in the system and how many code clones occur across all versions of the software system.

In the next section, we describe the final design of CloneCompass to hopefully address these challenges. As mentioned, our design work also helped us clarify the challenges we described above.

V. Approach

To support the exploration of scalable many-to-many comparisons, the first two challenges we discussed above, we reviewed previous research concerning visualization techniques for large-scale data that shows an overview first and then provides details on demand [12] [13]. Building on this research, we propose CloneCompass, a tool with two visualizations that each target a different level of abstraction: 1) an aggregated TreeMap Matrix that provides an overview of the distribution of the entire dataset of clones across one or more systems, and 2) a fine-grained 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.

A. Design Process

As we mentioned in Section II, matrix-based views have been used to show comparisons between different software systems, versions, files, etc. [7] [8] [9]. Inspired by these studies, we 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 limits—users might not gain useful information when a large number of data are simultaneously shown on the screen. Through our literature review, we found that previous studies [18] [19] used aggregation techniques to reduce the amount of data shown. In an attribute-based aggregation technique, nodes in a graph are aggregated into groups by an attribute, and each group is treated as an aggregated node [21]. In our case, nodes are functions ($n_0, n_1, ..., n_m$ in Fig. 4 (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. For the categorical attributes (binary and version), the functions from the same binary or version are aggregated into the same group. For the quantitative attributes (code size and block size), after the functions are ordered by code size or block size, a binning process is used to divide the ordered functions into distinct ranges, and the functions in each range are then considered as a group. Which attribute is used for ordering and aggregation is set by the user.

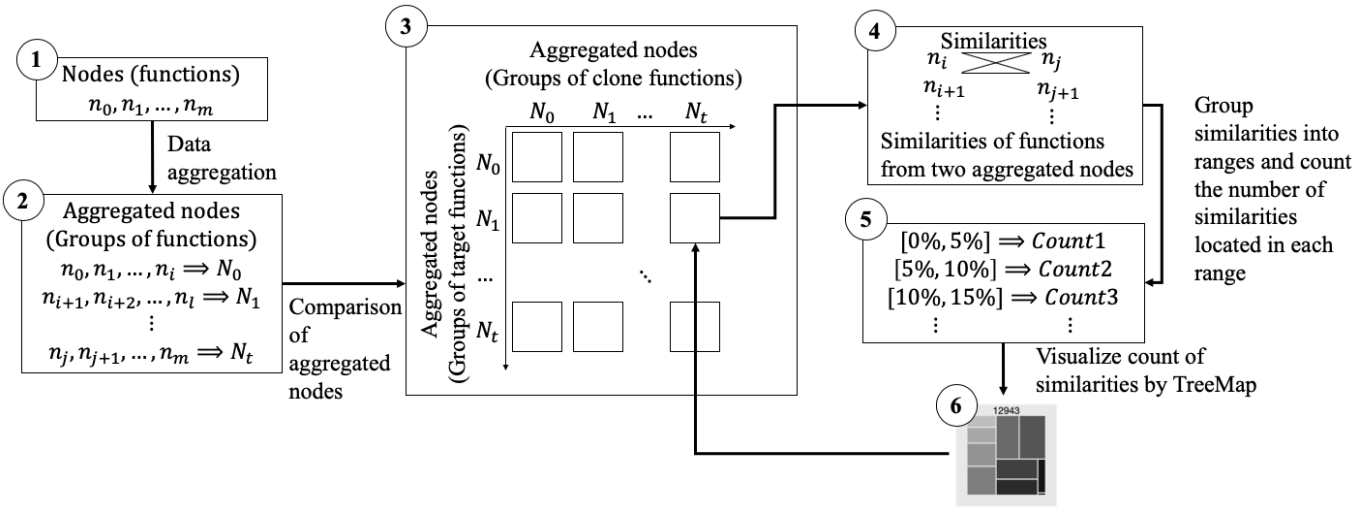

Fig. 4. The design process

After aggregation, each aggregated node is a group of functions ($N_0, N_1, ..., N_t$ in Fig. 4 (2)). If we place aggregated nodes at the rows and columns in a matrix-based view (Fig. 4 (3)), each node in a row represents a group of target functions, and each node in a column represents a group of clone functions. The intersection of a row and a column indicates a group of target functions and a group of clone functions, and the similarities between target functions and clone functions can be shown in the intersection (Fig. 4 (4)).

Since the similarities could range from a very low value (two functions are different) to a very large value (two functions are similar), splitting the similarities into distinct ranges can help the user focus on specific similarity ranges. We split the similarity ranges with a step of 5% and calculate the number of function-clone pairs having similarities located in each similarity range (Fig. 4 (5)). For example, X function-clone pairs have similarities from 75% to 80%.

Next, we focus on what visual representation can be used to show the similarities of the functions from two aggregated nodes. After trying several visual representations, we decided to use the TreeMap as it performs well, both in expressiveness and effectiveness, and it saves space. As shown in Fig. 4 (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), and 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.

B. TreeMap Matrix

By placing the TreeMap at each intersection of a row and a column in Fig. 4 (3), a TreeMap Matrix is formed. From the TreeMap Matrix, users can see how the similarities of functions are distributed in the whole dataset. When the user hovers over an outer or inner rectangle in a TreeMap, a blue border indicates which rectangle is being inspected, and a tooltip next to the rectangle shows how many function-clone pairs this rectangle contains.

If functions are ordered and aggregated by a quantitative attribute (code size or block size), the left and top nodes represent functions with smaller code size or block size. Each TreeMap shows the similarities of both clone and target functions with a code/block size within a certain range. For example, in Fig. 5 (a), the TreeMap with the blue border shows the similarity distribution of target functions with a code size of 52 to 119, and clone functions with a code size of 20 to 51. When hovering over the outer rectangle of the TreeMap, a tooltip shows 1,106 function-clone similarities in the range of 75% to 100%. Similarly, if functions are aggregated by a categorical attribute (binary and version), each TreeMap shows the similarities of functions from two binaries or two versions. For example, in Fig. 5 (b), four versions of a binary are shown in the TreeMap Matrix, and each TreeMap shows the similarities between two versions. In each TreeMap, the rectangles show the similarities within a specified range. For example, in Fig. 5 (b), when hovering over the rectangle with the blue border, a tooltip shows the rectangle contains 1,491 similarities in the range of 95% to 100%. The Treemap Matrix provides the following two design features to assist the user.

1) 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, shown in Fig. 6). 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, we calculated the Kam1n0 search results of 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—

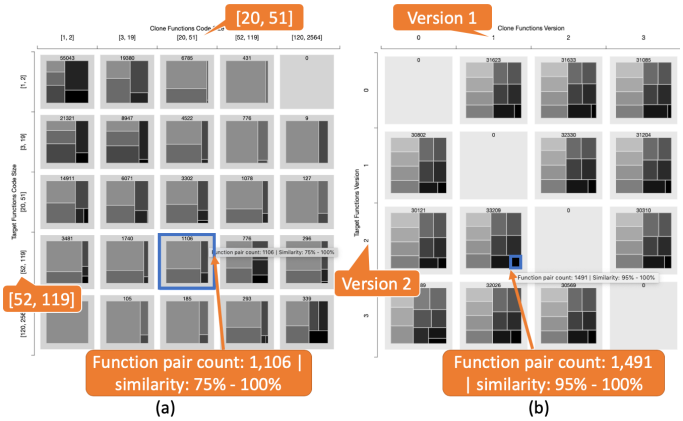

Fig. 5. TreeMap Matrix: (a) ordered and aggregated by code size, while hovering over a TreeMap; (b) aggregated by version, while hovering over a rectangle

pagination overcomes the screen size limitations and ensures all TreeMaps are readable.

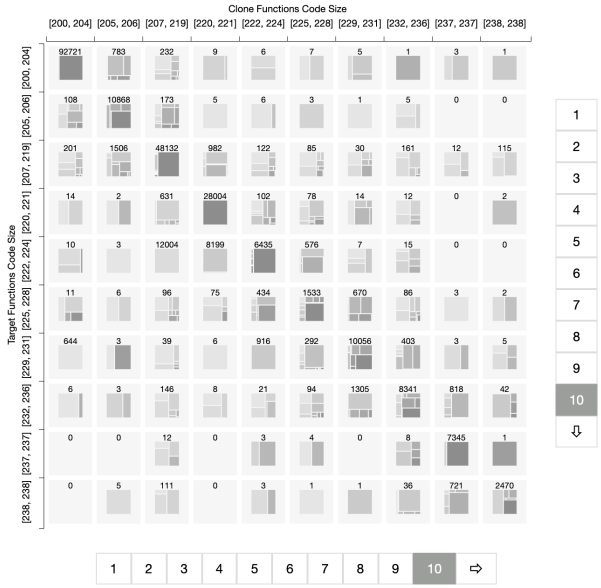

Fig. 6. TreeMap Matrix with pagination

2) 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 [14] [15] [16] 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, we use multiple selections to allow users to see the detail view of the data from any area: the user can select multiple outer and inner rectangles for a given Treemap (see Fig. 7). By clicking an outer rectangle, all functions from the two aggregated nodes of the TreeMap can be selected. By clicking an inner rectangle, the functions from the two aggregated nodes of the TreeMap with a *specified similarity* range can be selected.

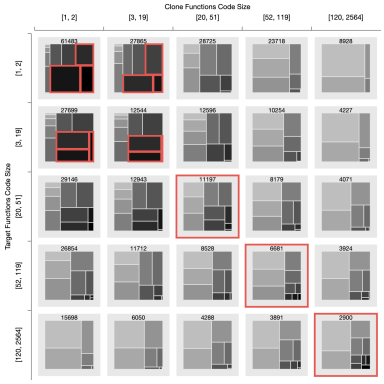

Fig. 7. Multiple selections of inner and outer rectangles in the TreeMap Matrix

C. Adjacency Matrix

After rectangles are selected in the TreeMap Matrix, a comparison of functions from these rectangles is shown in an Adjacency Matrix view. In the Adjacency Matrix, the row and column are target and clone functions ordered by the same attribute 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.

1) Zooming and panning: The number of function-clone pairs shown in the Adjacency Matrix varies by the selected rectangles. To ensure that 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/panning is shown in Fig. 8. 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 next to the Adjacency Matrix shows more detailed information including what binaries the functions are from.

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 select various sized datasets and choose appropriate or preferred glyphs and colors.

Two glyphs, square (Fig. 8) and FatFont [22](Fig. 9), are applied to represent the similarity in each cell of the Adjacency Matrix. An experiment was conducted to compare the

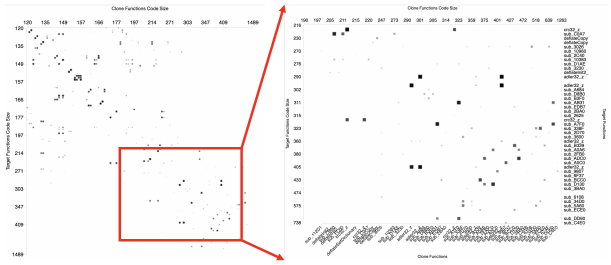

Fig. 8. Adjacency Matrix before and after zooming/panning

(a) (b)

Fig. 10. Color scheme choices and filters in the Adjacency Matrix

effectiveness of different glyph usage in Adjacency Matrices with weighted edges [23]. The authors showed that square and FatFont are good 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 square glyphs are more useful. When using square glyphs, different color

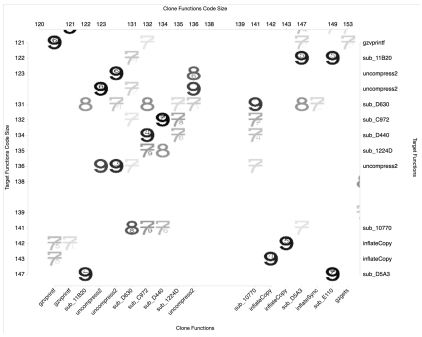

Fig. 9. FatFont in the Adjacency Matrix

schemes (see Fig. 10 (a)) 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 we applied the Adjacency Matrix to large-scale datasets, we 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[5], and a categorical color scheme, Viridis[5].

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.

3) Filters: Functions in the Adjacency Matrix are ordered by the aggregation attribute set by the user. Filtering allows a user to take other attributes rather than the aggregation attribute into account. By using filters (shown in Fig. 10 (b)), users can see multiple function-clone pairs constrained by differing 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.

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

D. Workflow

CloneCompass was implemented on a visualization development platform we created, Lodestone[6], that allows users to assemble 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 described as follows:

1) **Preprocess data:** The user inputs Kam1n0 search results into a preprocess script to calculate the data needed in CloneCompass.

2) **Load dataset:** The user then inputs preprocessed 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.

5) **Exploration:** 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 entire dataset by switching between these two matrix-based views.

VI. Preliminary Evaluation

As part of our design study, we conducted a preliminary evaluation with our stakeholders using CloneCompass. Our approach was to ask them to use CloneCompass to explore the clones in two given datasets: 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 consisting 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 with a collection of zlib libraries was generated

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

based on search results of demo binaries from Kam1n0's GitHub repository[7]. The second dataset was provided to us by our stakeholders and helped us demonstrate the scalability of our approach. These different datasets are each interesting to explore in terms of clones and potential vulnerabilities. Additionally, we encouraged the users to load their own datasets.

The evaluation was performed by two of Kam1n0's creators. One participant used the two datasets we provided, and the other participant used their own dataset of binaries that contained known vulnerabilities.

As our participants explored clones within and across these software systems, we asked them to use a "think-aloud protocol" [20] where the participants were encouraged to verbalize their thoughts while using CloneCompass.

Rather than specifying user tasks for this evaluation, we asked the participants to list questions they may pose about clones and vulnerabilities before using CloneCompass. This enabled us to 1) refine the challenges we captured previously (described in Section IV), and 2) identify potential challenges we had not identified earlier in our work.

A. Procedure

Tool exploration was remotely observed and recorded via screen sharing and 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) For the participant that used their own data (search results from Kam1n0), this participant ran a data pre-processing script before the interview. This script populated an SQLite3 database with consumable results usable within CloneCompass. For the participant that used their own demo data, this step was skipped as we sent them the SQLite3 database.
  2) Participants listed questions they expected to answer before 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 observation (1 hour):
  1) We 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 thought processes and asking questions when necessary. We interfered minimally and assisted if progress halted or if questions were asked.
- We conducted a feedback session where we asked 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 by using CloneCompass. We also asked user experience questions: e.g., are the two matrix-based views easy to read and use, and is CloneCompass user friendly.

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

B. Findings

Our findings are based on participant descriptions during the exploration of CloneCompass and the feedback they provided.

1) Challenge 1: Many-to-many comparisons: We found that both the TreeMap Matrix and the Adjacency Matrix performed well for many-to-many comparisons. For example, 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, participants were able to compare multiple functions and see clusters of clones. For example, by loading multiple vulnerable binaries in CloneCompass, a participant identified clusters of vulnerabilities. By exploring these clusters, the participant could then identify potentially undiscovered vulnerabilities that were grouped alongside known vulnerabilities.

2) Challenge 2: Scalability of clone exploration: We originally proposed the TreeMap Matrix view to overcome the scalability limit of Adjacency Matrices. Based on our 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. However, there is a limit to the number of TreeMaps that can be presented. For example, one participant's dataset contained 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.

3) Challenge 3: Big-picture understanding of how clones are dispersed across many systems: For this challenge, we found that the 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 could help 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"*. A 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"*.

4) Refining user requirements: In our evaluation, the participants found that challenges 1 and 2 were satisfied by our solution. Our interpretation of the third challenge was correct because the participants expected to see the distribution of clones. However, the questions we identified and prioritized, such as *finding smallest/biggest clones*, were not the partic-

ipants' focus. Instead they concentrated on target and clone functions with similar code sizes or block sizes.

We also discovered a new question that was not previously identified by us 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 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). The participant concluded that highly similar clones shared similar block sizes rather than similar code sizes in this dataset—in other words, clones tended to have similar block sizes.

In addition, one participant suggested that a user-defined aggregation attribute could be introduced in CloneCompass to compare the clones based on this new attribute. Specifically, during preprocessing, this would let the user write a regular expression defining how to extract an attribute 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. Also, efficient interactions between Kam1n0 and CloneCompass could be useful for our stakeholder's workflows. 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 the user perform more detailed and exploratory analyses of clones.

5) User experience: The user experience was validated through our observations of the participants using CloneCompass and the questions asked in the interviews. In general, CloneCompass was seen as *"user friendly"* and participants did not have many difficulties using it (according to their feedback). However, since the visualizations contained 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"*. We summarize the pros and cons about user experience below.

a) TreeMap Matrix: The participants reported that the TreeMap Matrix was easy to read and understand. One participant stated that the initial understanding of the number on the top of each TreeMap could benefit from the provided tooltip. One participant particularly liked the multiple selection functionality. However, we 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"*.

b) Adjacency Matrix: The 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, we received feedback on the glyphs used in our Adjacency Matrix: one participant remarked that *"FatFont is easy to use because you can see the actual value"*.

c) Interacting across two matrix-based views: CloneCompass allows users to switch between two matrix-based views, through which one can explore the whole dataset. However, when switching between the two views, one participant was concerned because they forgot what had already been selected in the TreeMap Matrix. This is a meaningful insight: an annotation technique or browsing history would help users remember what data was 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 square color 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 used might be necessary.

In this preliminary evaluation, we found that CloneCompass helped participants address the three challenges characterized in Section IV. We also discovered questions CloneCompass could answer that had not previously been recognized, and were able to refine user requirements and better understand the user experience through participant feedback. In the next section, we focus on the implementation and performance measurements of CloneCompass.

VII. Implementation and Performance Measurements

As mentioned in Section V, CloneCompass was implemented on a visualization development platform, Lodestone[8], which is built in Typescript[9] and uses React.js[10], Pixi.js[11] and D3.js[12]. SQLite3[13] is used to save preprocessing results as it is a file-based dataset engine that is easily transferred to users. In Lodestone, we deployed CloneCompass to participants as an Electron[14] app along with our SQLite3 database.

The performance of CloneCompass was measured on an Intel Core i7 CPU @ 3.40 GHz computer with 16GB of RAM and an NVIDIA Quadro 600 graphics card. Two datasets were used in the measurement: 1) a small dataset containing a collection of zlib libraries compiled with different optimization techniques; 2) and a large dataset containing a collection of browser binaries, including Chromium, Opera, Firefox, SRWare Iron, Comodo Dragon, and two other binaries (a libpng and a zlib library).

The small dataset included 532,730 function-clone pairs and 899 functions. The pre-processing took less than one minute,and there was no noticeable latency for loading and interactions.

[8]https://thechiselgroup.org/lodestone/
[9]https://www.typescriptlang.org/
[10]https://reactjs.org/
[11]http://www.pixijs.com/
[12]https://d3js.org/
[13]https://www.sqlite.org/index.html
[14]https://electronjs.org/

The large dataset included 94,082,393 function-clone pairs and 873,626 functions. The pre-processing step took about 2.5 hours. The loading time of a TreeMap Matrix aggregated by code size, block size, or binaries was about 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 3 seconds to load 7,486 function-clone pairs and 6 seconds to load 20,212 function-clone pairs in the Adjacency Matrix.

VIII. Discussion

CloneCompass uses a novel TreeMap Matrix view to show a large graph of comparisons between aggregated node pairs. Compared with other matrix-based views [14] [15] [16] [18] [19], a TreeMap Matrix has three strengths. Firstly, in contrast with the visual representations (e.g., color shade) of comparisons between aggregated node pairs, TreeMaps are two-dimensional views containing rich information. Secondly, 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. Thirdly, 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. Compared with multiple selections, the zooming techniques used in existing studies can only allow the user to see the data from a limited area in a detailed view. The use of multiple selections addresses the space limitation and allows the user to choose and zoom in on any area.

In our study, we found that a user can efficiently explore datasets and gain insights into code cloning by using CloneCompass, which includes both a TreeMap Matrix and an Adjacency Matrix. The combination of the two matrix-based views enables users to explore any and all 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 across long time periods; and 5) comparing clones of multiple binaries or versions. Such insights help users understand how clones are dispersed across many systems and accomplish goals like identifying potential vulnerabilities.

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

In our study, the users identified some improvements to the visualizations that would help them achieve their goals. Users tend to forget what they have visited after several switches between the TreeMap Matrix and the Adjacency Matrix. An annotation technique or browsing history would help users remember what they have visited. As well, when the number of pages shown in the TreeMap Matrix is too large, it is not easy to find a suitable combination of two pages at the bottom and on the right of the TreeMap Matrix. Also, 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 currently we do not know the maximum it can handle because of our limited access to larger datasets. As a result, it is possible that a TreeMap Matrix aggregated by binaries could not be loaded when the number of binaries is too large. Besides, since the Adjacency Matrix shows a subset of function-clone pairs from a TreeMap Matrix, when the selected rectangles in the TreeMap Matrix contains a large number of function-clone pairs, the Adjacency Matrix could take a very long time to display. The scalability limits of the TreeMap Matrix and Adjacency Matrix can be improved by optimizing CloneCompass' code performance.

IX. Conclusion and Future Work

In this paper, we report on a design study to support assembly code clone analysis. We characterized the challenges experienced when users explore large-scale search results from Kam1n0, an assembly code clone search engine. We then iteratively designed and refined a novel interactive visual tool, CloneCompass, and then conducted a preliminary evaluation as part of our design study.

CloneCompass pairs a TreeMap Matrix view with an Adjacency Matrix view to show and support the exploration and inspection of a large number of assembly code clones from the search results of Kam1n0. The evaluation results show that users can gain useful insights while using CloneCompass during their exploration of assembly code clones. User insights include understanding the software ecosystem's clones and finding further software vulnerabilities. A video demo of CloneCompass has been uploaded to the following link https://youtu.be/IIcefibBalI.

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

Acknowledgments

We would like to thank Martin Salois, Steven H.H. Ding, and Benjamin C.M. Fung for their great advice and support. We would also like to thank Matthieu Foucault who provided insight and expertise that greatly assisted the research. We also thank Cassandra Petrachenko for comments that improved the manuscript.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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.

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

[16] 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.

[17] N. Henry and J.-D. Fekete, "Matrixexplorer: a dual-representation system to explore social networks," *IEEE transactions on visualization and computer graphics*, vol. 12, no. 5, pp. 677–684, 2006.

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.