International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 6– No. 6, December 2013 – www.ijais.org

Lattice-based Metaphor for Visualizing Disassembled Executable Code

Peter Mulwa
School of Computing & Informatics
University of Nairobi
Nairobi, Kenya

Tonny Omwansa, Ph.D
School of Computing & Informatics
University of Nairobi
Nairobi, Kenya

ABSTRACT

Lattice-based structures provide a means of encoding information. This inherent property of information representation is utilized to design a metaphor for visualizing and analyzing a program, based on the structured nature of disassembled executable code.

Beginning from a generic platform's Instruction Set Architecture (ISA) and abstracting the manner in which instructions are combined to form a program, a generic representation of the flow of a program is created. This representation is then mapped onto a lattice-based structure for visualization. Once the visualization is rendered, the lattice structure is used to analyze a program's disassembled code in order to extract potentially useful information for decision making.

General Terms

Binary Code, Visualization, Instruction Set Architecture

Keywords

Lattice, Metaphor

1. INTRODUCTION

In varied scenarios, executable files need to be reverse engineered in order to understand their functionality. Disassembling an executable provides a human-readable format that resembles the underlying machine code due to the one-to-one mapping of machine and assembly code. Dependent on the size of the executable, the quantity of the information generated can be large. This makes the analysis of information potentially difficult. Besides textually viewing the content, visualization can be utilized to enhance the process of understanding and analyzing the content.

Lattices provide a potentially useful structure that can be adapted to develop a visual metaphor that can be used to visualize & analyze a program's disassembled executable code in order to generate usable information to aid in decision making.

This paper presents a development of a lattice-based metaphor for this purpose. It begins by abstracting a generic platform's Instruction Set Architecture (ISA). Rules are then formulated on how to represent the different combination of instructions in order to enable adaptation to a lattice structure. A notation for displaying information is developed. Various basic code constructs dealing with branching and looping are then illustrated by a process of abstracting their structural design and then visualizing them using the metaphor on the basis that these constructs are combined in various ways to constitute a program.

2. REVIEW OF RELATED LITERATURE

Software Visualization provides an alternative means of viewing programs beyond textual representations. In visualizations metaphors represent different aspects of code. However, since code is abstract, these metaphors can take various forms such as geometric shapes [1][2] or real world objects [1]. In addition to shape, other visual attributes include size, height/depth, colour, texture/bumpmaps [3], transparency, elevation, and position. These represent various code attributes such as sequence, control structure, nesting level, declarations and implementations, classes and inheritances, etc. [3][4][5][6][7].

Various representations have been proposed, such as pixel maps and cylinder bars [4], matrices and rows-columns [5], treemaps and edge bundling [2], treemaps [8][9], hulls [7], radial [8], kiviat [9], and cartographic [10]. The representations could be used concurrently [11][12] providing different views of the same or different information.

Visualizations are utilized to generate usable information. As information to be analyzed increases, new ways to analyze information is required. 3D visualization is being utilized to enhance existing metaphors, for example, pixel maps [4], kiviat [9], hulls [7], and edge bundling [6]. 2D view scalability is hindered as content increases [5], and even with zoom [11] or multiple views [8][12] features, they are prone to cognitive overload and lack of intuitiveness [13]. Extending visual analysis to 3D enables increasing the spatial space available for interacting with information with the benefit of adding a new spatial dimension [1], enhance memory activity [4], and ease analysis of information [14].

Visualization's goal is to increase the level of understanding of the information being processed, possibly by maintaining a consistent mental model [14] for recurrent use [10]. Richard Hamming's statement, 'insight, not number is what computing should evolve to', is a guiding principle. Abstraction of complex aspects to everyday equivalents [15], incorporating animations [9][15], lowering clutter by component aggregation [13], direct manipulation [14] help to increase understanding. Aspects such as navigation and location identification can be enhanced by limited animation [6] and panning features [14].

Use of Graphical Processing Units (GPU), from a rendering perspective [14], could be utilized to enhance performance, for example, with texturing, which is natively performed by a GPU.

Due to the varied potential uses of visualization and the abstract nature of the information, a methodology may be required to determine the ideal visualization for a given scenario. Two parameters namely, the data set (may require prototyping) and task analysis (with parameters such as overview, zoom, filter,

details-on-demand, relation, history, and extraction) can be used in the visualization design [6][14][16].
However, visualization is unique to the problem domain; if a methodology doesn't fit, the alternative could be either to modify the design, add functionality, or use different concurrent visualizations [14].

Visualization is applicable to the entire software lifecycle, including the support of legacy systems [1], in security analysis [16], usable for both the source & binary code, malware acquired in binary form [17][18], and source code from the perspective of metrics, classes & packages, whole software structuring, and whole software porting respectively [4][5][8][12]. With refactoring [11] effort estimation and rewrite impact can be determined; or with maintenance [12] identification of high code turnover areas for purposes of either rewriting, defect identification, regression tests, or fan in and fan out metrics can be determined. The visualizations can even be integrated with other tools via input and output files [10][12].

Abstraction plays a role in reducing information and cognitive overload. Two concepts that support this exist [1][4]. Elision property of 'abstract distant objects, detail closer objects', and Bruce Shneiderman's visualization mantra, which specifies the detail sequence of 'overview first, zoom & filter details on demand'. Complementing components for information extraction are visual and textual representations. Visualization is usable for higher levels and textual for lower levels [16]. The complex interaction between software entities are prone to make visualization cluttered with the potential effect of increasing the cognitive load [2][16] and ignoring information [10].

Enhancing abstraction is possible by not displaying all information at once. Pertinent information can be displayed dependent on the current context or upon demand by encoding it in the metaphor [8]. Furthermore, mental models can be utilized to aid in program comprehension. Conceptual and structural models and the concepts of anchors, can provide a reference point during analysis [10].

Two properties of metaphors are expressiveness and effectiveness [1]. Expressiveness refers to the capability to represent the required information, which enhances the display of concise information. Effectiveness refers to the visibility & idealness of the required information encapsulated and presented in the metaphor, which enhances cognitive processing. Both properties provide tools for the design and evaluation of metaphors [4].

The literature shows that various visualization undertakings have been done with software and its attributes for purposes of improving the understanding of programs from both the binary and source code levels. Binary code, however, would provide a more accurate form for analysis in reverse engineering, as it is what is actually executed on a computing device. The literature brings out the concern of information overloading during the analysis of large quantities of information. Various solutions are proposed and guidelines presented to address the concerns of software visualization.

3. METHODOLOGY

3.1 Instruction Set Architecture (ISA)

Given any executable file, its executable contents can be converted into an equivalent assembly program listing via a disassembler for the target platform. The ISA for a given platform describes the platform's assembly language programming interface. These instructions are utilized to generate programs by combining them together, possibly either at the assembler level, or using a higher level language.

At the ISA level of abstraction, the instructions can broadly be classified as either being sequential or control type instructions. Sequential type instructions are executed and the immediate following instruction then executed. Control type instructions, on the other hand, have the potential to alter control flow. The figure below illustrates this concept. Hence, executable code can be considered as comprising of different permutations of sequential & branching instructions using the available ISA.

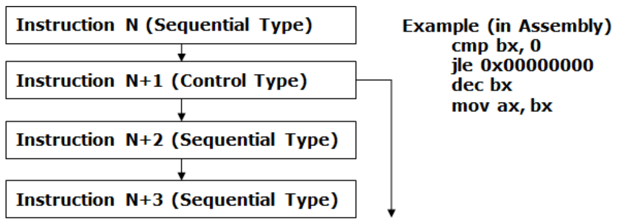
Instruction N (Sequential Type)
Instruction N+1 (Control Type)
Instruction N+2 (Sequential Type)
Instruction N+3 (Sequential Type)
Example (in Assembly)
cmp bx, 0
jle 0x00000000
dec bx
mov ax, bx

Figure 1: Conceptualized illustration of sample sequential & control type ISA instructions

3.2 ISA Visualization – Lattice Metaphor Evolution

In order to provide an optimized visualization of a program comprising of sequential & control type instructions, the sequential portions can be aggregated as illustrated in the figure below describing the generation of a lattice-metaphor for an example program flow of 10 instructions.

⬛	Sequential Instruction
🟦	Control Instruction (not Runtime Dependent)
▨	Control Instruction (Runtime Dependent)
⬜	Current Instruction (Current Focus)
🟥	Lower Branching Destination from Current Instruction
🟩	Higher Branching Destination from Current Instruction

Figure 2: Legend

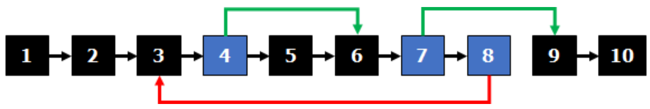

Figure 3: Sequential & Control Instructions

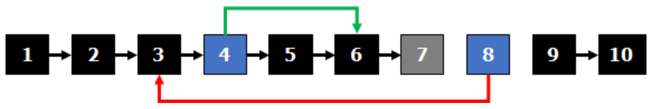

Figure 4: Instruction 7 is identified as runtime dependent (indicated by change from blue to grey notation)

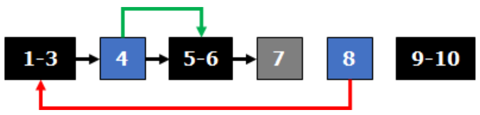

Figure 5: Aggregation of sequential instructions

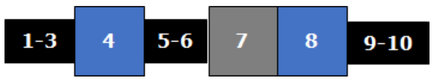

Figure 6: Elimination of flow arrows; branch information encoded within metaphor and control-type instructions differentiated by size

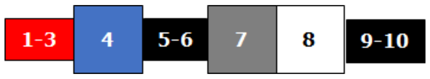

Figure 7: Illustration of encoded branch information for node 8 (white node) branching to lower address at node '1-3' (red node)

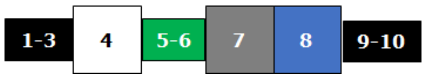

Figure 8: Illustration of encoded branch information for node 4 (white node) branching to higher address at node '5-6' (green node)

3.3 Folding Instruction Sequences into Sections

As the quantity of nodes representing instructions increases, the linear growth in the X-axis is limited by the available screen space. In order to accommodate the growth, the Z-axis is utilized. However, this requires the specification of a dimension.

A dimension refers to the number of nodes that will be displayed in the X-axis. For example, if the dimension is set to 10, then the 11th node will wrap around and be displayed at the next incremented Z-axis index. In order to maintain clarity of the wrapping, since the size of a node is the same along both the X-axis and Z-axis, a spacing equivalent to the size of the node is used to separate the different Z-axis indices. Consequently, the number of nodes along the Z-axis will be half the specified dimension.

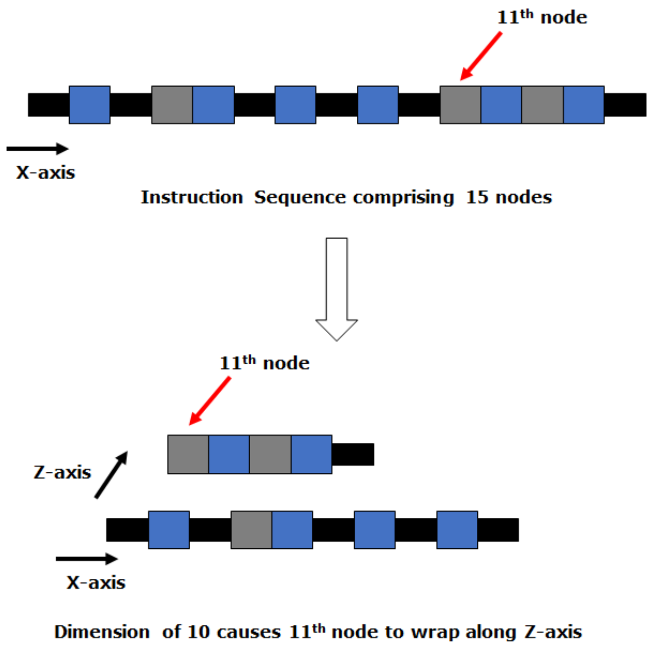
11th node
X-axis
Instruction Sequence comprising 15 nodes
11th node
Z-axis
X-axis
Dimension of 10 causes 11th node to wrap along Z-axis

Figure 9: Folding node sequences

3.4 Building Sections

Once the maximum number of nodes along the Z-axis is reached (based on a specified dimension), the resultant collection of nodes is referred to as a 'Section', which is represented by a differently notated node. The next node after a 'Section' is formed becomes the 1st node of the next 'Section'.

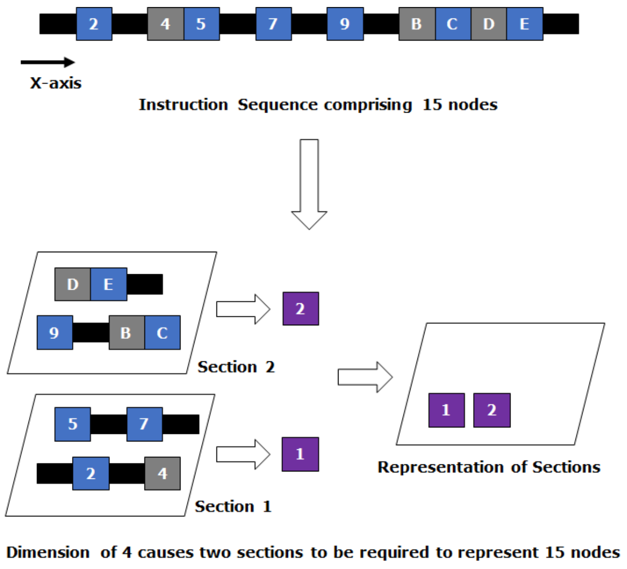
2
4
5
7
9
B
C
D
E
X-axis
Instruction Sequence comprising 15 nodes
D
E
9
B
C
Section 2
2
5
7
2
4
Section 1
1
1
2
Representation of Sections
Dimension of 4 causes two sections to be required to represent 15 nodes

Figure 10: Section Formation

As the number of sections increase, their layout is ordered along the axes in the following order: X-Z-Y. The dimension used when representing sections is defined by the number of sections, with the value being the cube-root of the number of sections.

3.5 Navigation of Nodes & Sections

The current location is identified by a white highlighted focus. When a node is selected any encoded information that is displayed is based on the node. Due to the multi-dimensional nature of the lattice-structure, navigation in the various domains is possible and sequential movement from one node to the previous or next is not mandatory.

Movement forward, along the X-axis, from the last node in a given section results in the next section being displayed and its 1st node highlighted, while movement backward from the 1st node in a given section results in the previous section being displayed and its last node highlighted. Movement along the Z-axis results in a move equivalent to the dimension, giving the effect of either an upward or downward movement. Any movement when analyzing the individual nodes, results is its equivalent section being highlighted in order to provide a node's reference in relation to the entire program.

Movement in the section resembles the above description of the nodes, with the difference that when a different section is selected, the 1st node of that section is the one highlighted. This is because a 'Section' represents more than 1 'Node'.

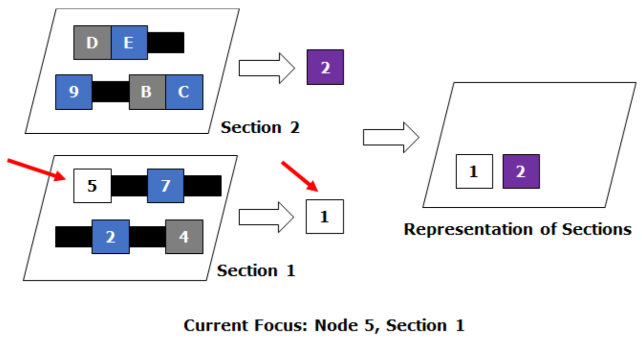

Figure 11: Initial selected node is 5

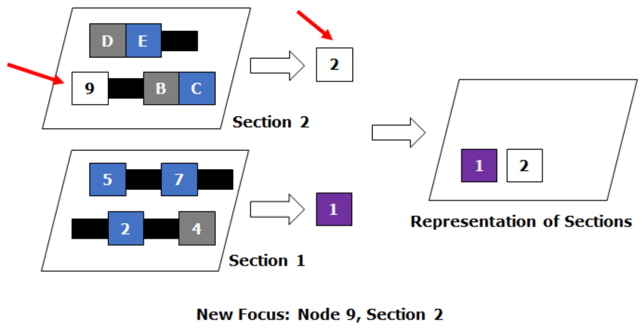

Figure 12: Location after navigation from node 5 to 9

4. DATA ANALYSIS AND RESULTS

4.1 Introduction

The process of visualization and analysis begins from the acquisition of an executable file with its associated disassembler for the given platform. This enables the visualization tool to be platform-independent. Once the executable file has been disassembled in an assembly listing, the content of the file is parsed and then imported into the visualization application prior to visualization.

4.2 Test Data

A test program was implemented to test the visualization and analysis capabilities of the lattice-based metaphor. In order to easily identify these constructs at the assembly level, compiler optimizations were disabled. The C/C++ programming language is used to write the program. The Intel x86 ISA is utilized at the assembly language level. The disassembler utilized is the dumpbin.exe file included with the Microsoft Visual Studio Integrated Development Environment.

Source Code Listing of test program

```
int Add(int n1, int n2);
int main() {
        int nSum, nCount , nCondition, nValue;
        nCount = 10;
        nCondition = 1;

        for (int i = 1; i < 11; i++)
                nSum += i;
        Add(nSum, 2);
        do {
                nCount--;
        } while (nCount);
        if (nCondition) {
                nValue++;
        } else {
                while (1) {
                }
        }
        return 0x1234;
}
int Add(int n1, int n2) {
        return n1 + n2;
}
```

4.3 Visualization

Once the disassembled code was loaded into the application, the following visualization was generated.

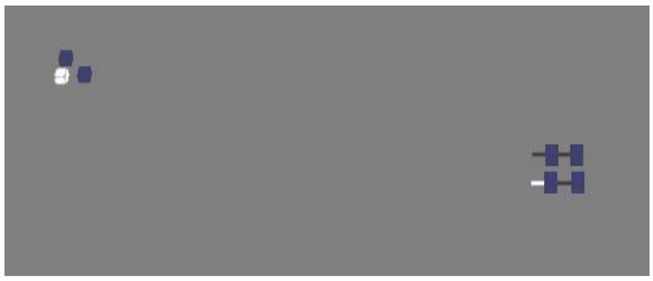

Figure 13: Overall Program Visualization (Dimension set to 4 gives 3 sections; 1st node and section highlighted)

4.4 Analysis

This section discusses the following visualization analyses:

- Navigation – next location highlighting
- Potential source locator

4.4.1 Navigation – Next Location Highlight

The structure enables non-linear navigation of the visualization and identification of overall as well as branching location (to either a higher offset indicated by a green highlight, or to a lower offset indicated by a red highlight) based on the current location (indicated by a white highlight.

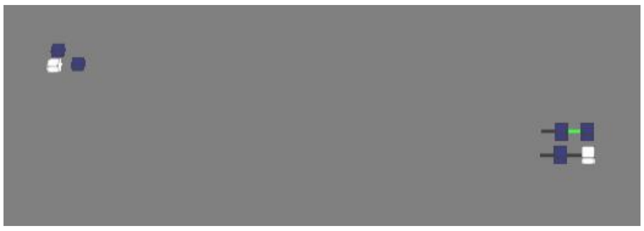

Figure 14: Next location highlight; bypassing 'for' loop

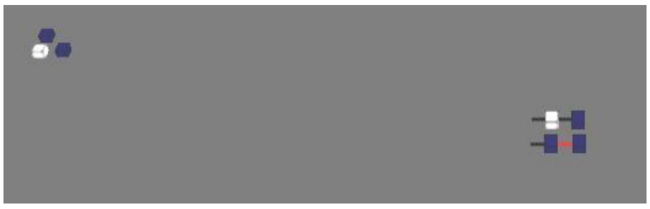

Figure 15: Last instruction in 'for' loop transferring control back to beginning of loop

4.4.2 Potential Source Locator

Navigation through a program's flow is usually in a forward direction, i.e. from the current location to potential next locations (yellow highlight) either sequentially or by control branching. However, the capability of being able to identify potential areas that could have resulted in a branch to the current location is beneficial.

The feature is implemented by searching for instructions that contain the selected node as the destination address. The offset of the identified instructions are then identified as potential source locators.

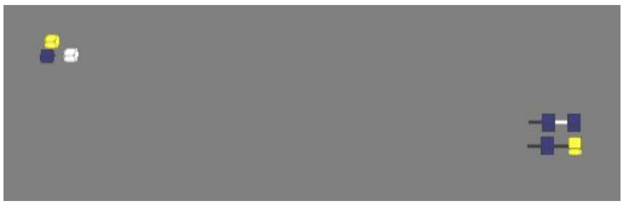

Figure 16: 'while' loop entry reachable from 2 different locations

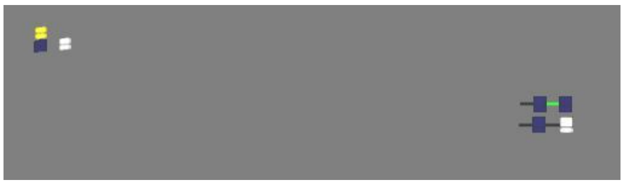

Figure 17: 1st potential source location from bypassing 'if' statement or running through

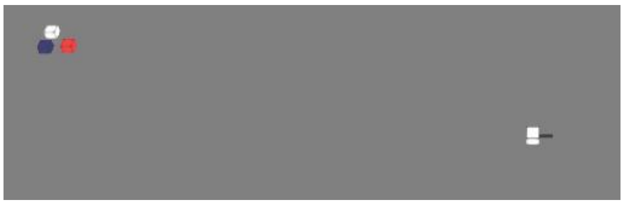

Figure 18: 2nd potential source location from 'while' statement (via the jump back to the start of the loop)

5. CONCLUSION

As visual processing capability increases with enhancements in GPUs, visual analysis of information is potentially possible. This paper explored a lattice-based metaphor for visualizing & analyzing disassembled executable code. The feasibility of a lattice structure for both visualization & analysis was illustrated as one possible alternative in software visualization. Code constructs visualization was shown to be feasible. In addition, various analyses of disassembled code as well as statistical information generation were illustrated. The research also created a potential framework for the design of new metaphors via abstracting the underlying concepts, generating the basic building blocks, developing a notation, and finally designing an interaction mechanism.

The potential of code obfuscation and compiler optimizations exist when dealing with software, which could make the disassembly and analysis difficult. The research assumed the absence of both. This provides an area for further research work.

Various areas of enhancements exist. Currently, the metaphor relies on only 2 node models – for sequential and control type instructions. However, for control instructions various models could be used to indicate the type of control instruction as well as the direction of branching. GPU features such as lighting could also be used to enhance the metaphor. The potential of being able to drag nodes in and out of the metaphor would provide another level of visually manipulating and interacting with programs.

6. ACKNOWLEDGEMENTS

We would like to thank Dr. Wausi, Dr. Nganga, and Dr. Orwa for their input during the development process.

7. REFERENCES

[1] Grancanin, Denis, et al (2005), Software Visualization, Innovations in Systems and Software Engineering, September 2005, Volume 1, Issue 2, Pages 221-230

[2] Caserta, Pierre, et al (2011), 3D Hierarchical Edge Bundles to Visualize Relations in a Software City Metaphor, VisSoft 2011 IEEE International Workshop on Visualizing Software for Understanding and Analysis, September 2011

[3] Holten, Danny, et al (2005), Visual Realism for the Visualization of Software Metrics, IEEE Workshop on Visualizing Software for Understanding and Analysis, 2005

[4] Marcus, Andrian, et al (2003), 3D Representations for Software Visualizations, SoftViz 2003 ACM Symposium on Software Visualization, 2003

[5] Zeckzer, Dirk (2010), Visualizing Software Entities Using a Matrix Layout, SoftViz 2010 ACM Symposium on Software Visualization, 2010

[6] Beck, Fabian, et al (2011), Visually Exploring Multi-Dimensional Code Couplings, VisSoft 2011 IEEE International Workshop on Visualizing Software for Understanding and Analysis, September 2011

[7] Lambert, A, et al (2012), Visualizing Patterns in Node-Link Diagrams, 2012 International Conference on Information Visualization, July 2012

[8] Reniers, Dennie, et al (2011), Visual Exploration of Program Structure, Dependencies, and Metrics with Solid

SX, VizSoft 2011 IEEE International Workshop on Visualizing Software for Understanding and Analysis, September 2011

[9] Kerren, Andreas, et al (2009), Novel Visual Representation for Software Metrics using 3D and Animation, Software Engineering Workshop, 2009

[10] Kuhn, Adrian, et al (2010), Embedding Spatial Software Visualization in the IDE: An Exploratory Study, SoftViz 2010 International Symposium on Software Visualization, 2010

[11] Broeksema, Bertjan, et al (2011), PortAssist: Visual Analysis for Porting Large Code Bases, VizSoft 2011 IEEE International Workshop on Visualizing Software for Understanding and Analysis, September 2011

[12] Maletic, Jonathan I., et al (2011), MosaiCode: Visualizing Large Scale Software (A Tool Demonstration), VizSoft 2011 IEEE International Workshop on Visualizing Software for Understanding and Analysis, September 2011

[13] Holy, Lukas, et al (2012), Lowering Visual Clutter in Large Component Diagram, 2012 International Conference on Information Visualization, July 2012

[14] Wiss, Ulrika, et al (1998), Evaluating Three-Dimensional Information Visualization Designs: A Case Study of Three Designs, IEEE Conference on Information Visualization, July 1998

[15] Medani, Dan, et al (2010), Graph Works – Pilot Graph Theory Visualization Tool, SoftViz 2010 ACM Symposium on Software Visualization, 2010

[16] Goodall, John (2009), Visualization is Better! A Comparative Evaluation, VizSec 2009, International Workshop on Visualization for Cyber Security, October 2009

[17] Quist, Danny, et al (2009), Visualizing Compiled Executables for Malware Analysis, VizSec 2009, International Workshop on Visualization for Cyber Security, October 2009, Pages 27-32

[18] Trinius, Philipp, et al (2009), Visual Analysis of Malware Behaviour Using Treemaps and Threaded Graphs, VizSec 2009, International Workshop on Visualization for Cyber Security, October 2009, Pages 33-38