Program Comprehension Support for Assembly Language:
Assessing the Needs of Specialized Groups

by

Jennifer Ellen Baldwin
B.Sc., University of Victoria, 2004
M.Sc., University of Victoria, 2006

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

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Program Comprehension Support for Assembly Language:
Assessing the Needs of Specialized Groups

by

Jennifer Ellen Baldwin
B.Sc., University of Victoria, 2004
M.Sc., University of Victoria, 2006

Supervisory Committee

Dr. Yvonne Coady, Supervisor
(Department of Computer Science)

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

Dr. Alex Thomo, Departmental Member
(Department of Computer Science)

Dr. Stephen W. Neville, Outside Member
(Department of Electrical and Computer Engineering)

Supervisory Committee

Dr. Yvonne Coady, Supervisor
(Department of Computer Science)

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

Dr. Alex Thomo, Departmental Member
(Department of Computer Science)

Dr. Stephen W. Neville, Outside Member
(Department of Electrical and Computer Engineering)

ABSTRACT Advances in software engineering and programming languages have had an impact on productivity, time to market, comprehension, maintenance, and evolution of software. Low-level systems have been largely overlooked in this arena, not only because of their complexities, but also the "bare bones" culture of this domain.

This dissertation investigates the program comprehension needs of two stakeholder groups using different assembly languages: a mainframe development group and a malware analysis group. Exploratory interviews and surveys suggest that the groups' needs may be similar at a high-level. However, a detailed study involving requirements elicitation and case studies, reveals that the truth is much more complicated.

As a proof of concept, we have created the AVA (Assembly Visualization and Analysis) framework, which is independent of the underlying assembly language. Despite this independence, tools within AVA could not be applied with equal efficacy, even just within these two groups. This dissertation shows that there exist fundamental differences not only in the highly-specialized nature of each group's work, but also in the assembly languages themselves. This reality necessitates a disjoint set of tools that cannot be consolidated into a universally applicable framework.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to thank:

CA, for providing developers' time for interviews, feedback, and requirements elicitation.

DRDC, for contributing the ideas that led to winning the Hex Rays Best Plugin award in 2011, as well as the continuous testing and feedback from the team.

NSERC, for providing the resources that allowed me to complete this work.

DEDICATION

This dissertation is dedicated to my father. Without him, I would never have begun this journey.

I think back on how he tutored me every Sunday in Physics, an event I never looked forward to. But it was not just my Sunday, it was also his he gave up. My education was always so important to him.

I will always truly appreciate his encouragement and devotion.

Chapter 1

Introduction

Program comprehension is complex and time-consuming, particularly in manually tuned, low-level system codebases such as those written in assembly language. The current lack of adequate tool support for these systems further exacerbates this problem. Whereas engineers of higher-level systems quite often rely on tools for effectively navigating codebases and analyzing design, corresponding support for lower-level systems is severely lacking.

Software engineering and programming language evolution have distanced mainstream developers from low-level languages, having a dramatic impact on productivity, time to market, comprehension, maintenance and evolution of software in general. Low-level systems have been largely overlooked in this arena, partially due to the complexities they offer and partially due to the inherent "bare bones" culture in this domain. We believe this perceived cultural resistance is in part due to the fact that tool support, as we know it today, was not available to developers who worked primarily with assembly language and therefore, adoption is not widely accepted.

Assembly language comprehension tools have the potential to aid developers in many of the same ways as their high-level counterparts. Increased comprehension, coupled with navigation and development tools, could enable easier, faster and more reliable implementation in mainframe software, and analysis of security threats in malware. Another important factor is ensuring continuity when an expert leaves the team. New generations of developers are accustomed to a certain level of tool support. By reducing the barrier to adoption and possibility eliminating the cultural resistance, these developers may be able to reap the benefits that comprehension tools have brought to higher-level languages.

1.1 Problem Space

This section introduces the two groups of industry stakeholders we investigate in this dissertation: mainframe developers, and malware analysts. We further introduce the two assembly languages used by each group: HLASM and x86, respectively. Finally, we identify several unique challenges present in assembly code comprehension in general, and conclude with an overview of the concrete research questions and chapters in this dissertation.

1.1.1 Stakeholders

The first stakeholder group is CA (formerly known as Computer Associates). CA is a large software corporation with many offices worldwide. They create systems software, some of which runs in a mainframe environment. Furthermore, some of these systems are written in IBM's High Level Assembler language (or HLASM) [1] which runs on the z/Architecture mainframe environment. To our knowledge, teams located in Canada, USA, Czechoslovakia, and Australia, employ individuals who specialize in HLASM software development. Of these teams, we were able to cooperate with those in both Czechoslovakia and Australia.

The second stakeholder group is Defence Research and Development Canada (DRDC) Valcartier. DRDC is an agency of Canada's Department of National Defence (DND) which provides knowledge and technology to other areas of government. There are nine centres making up DRDC, one of which is the military research station located in Valcartier, Quebec. Our stakeholders at DRDC Valcartier are members of a team that use reverse engineering to investigate malware and security flaws. They disassemble malware executables into the x86 assembly language for analysis.

1.1.2 Background of HLASM and x86

Low-level languages can be of two types: machine languages and assembly languages [2]. These low-level languages are either machine code, or very close to machine code, providing little abstraction from a computer's instruction set architecture, and therefore the hardware. No compiler, or interpreter, is needed for such languages since the language itself maps directly to machine code. While these languages may be construed as "simple", they are in fact inherently complicated due to the intimate knowledge of the hardware architecture required to program effectively.

Assembly language is one step above machine language, in that it uses operands and operations, instead of binary digits, to compose a program. A program called an *assembler* then translates these slightly higher level instructions into machine language.

HLASM

High Level Assembler, or HLASM, is an assembly language created by IBM for its z/Architecture mainframe computers. The program structure differs greatly from most other assembly languages as it attempts to provide functionality that is usually only provided by high-level languages. In order to simulate the high-level approach to programming, certain structures not present in standard assembly languages were added to HLASM. There are two of interest in this dissertation, the DSECT and CSECT.

The dummy section (or DSECT) acts much like a struct in C and defines a set of data that is to be stored, but does not actually reserve any virtual memory for that data. The dummy section code is not translated into object code by the assembler.

Control sections (or CSECTs) are sections of code that are "independently relocatable" and can be moved to a different location in the assembly without affecting the functionality of the program. There are multiple assembler options that can be set that influence many aspects of control sections, such as the THREAD or NOTHREAD options. If THREAD is specified, the location counter is reset to zero at the beginning of each control section. Otherwise, the counter is continuous throughout the program unless it overflows, which would cause an assembler warning to be issued [3].

Since mainframes are specifically designed to support multiple users making large amounts of concurrent requests to the processor, the mainframe architecture is significantly more powerful than that of a standard personal computer. This adds to the complexity of the HLASM language.

x86

This assembly language is also known as the Intel 80x86 Series Assembly Language. The Intel 80x86 series consists of several generations of processors starting with the Intel 8086, which was released in 1978, and led to the current line of Pentium processors. The instruction set architecture, to this day, remains completely backwards compatible, although it has obviously been greatly expanded from the original 8086 in-

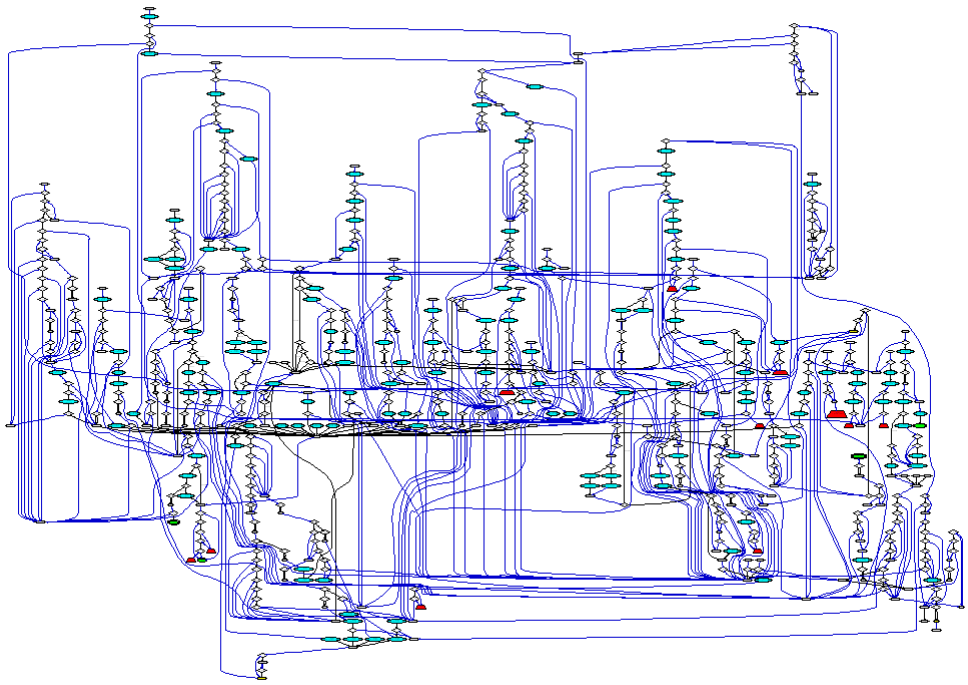

Figure 1.1: Complex Control Flow Graph.

struction set. This backwards compatibility has complicated the language somewhat since the original architecture was based on an 8-bit word, and was later expanded to support 16-, 32-, and 64-bit word architectures.

1.1.3 Challenges in Assembly Code Comprehension

Though program understanding has received much attention from the research community, these approaches and corresponding tools only have limited application to large scale low-level systems. Even fundamental characteristics such as control flow are exceedingly hard to track at scale in the systems we propose to target, as in those with excess of 100 KLOC and 10,000 branches. Unfortunately, assembly source code development leads to heavily optimized control flow with multiple entries and multiple exits. Additionally, the intertwining of multiple computation threads in a single source code module often results in complex control flow graphs for which decompositions are not easy, as shown in Figure 1.1 [4].

This is not meant to show bad programming structure, but instead a structure that is almost unavoidable in assembly programming. While this particular image is

provided by CA for HLASM systems, the same principle holds for applications written in other assembly languages. For example, analysts at DRDC have issues with the intentional obfuscation in malware, so that it is exceedingly difficult to pinpoint the security threat it contains. Consider the ways in which a typical call diagram is presented to developers in a state-of-the-art disassembler and debugger: IDA Pro [5]. Figure 1.2 shows a function call graph generated by executing the Mariposa botnet [6], and analyzing the memory dump.

One of the first things to note is that this is a static view that does not show an actual execution trace. The analyst cannot follow a call, see the ordering of the calls, or even know if a call occurs more than once—all of which are critical for comprehension. Additionally, this display is very limited in IDA Pro. For example, there is no way to locate a specific function; it has to be done by visually searching through the diagram.

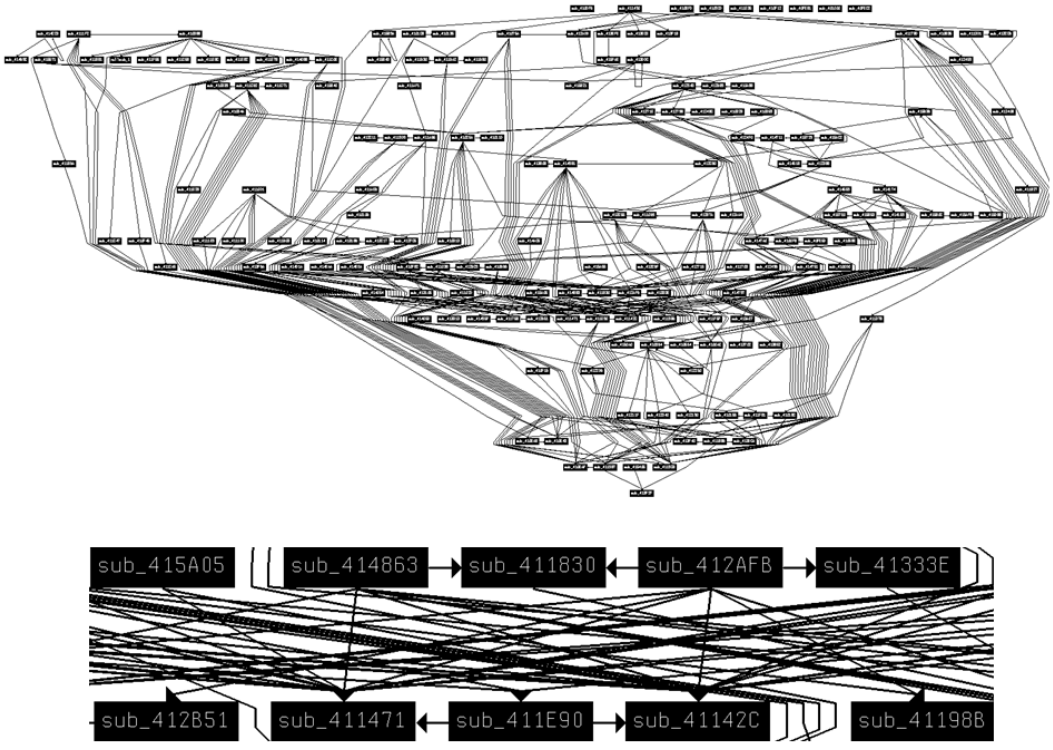

Figure 1.2: Mariposa Function Call Graph from IDA Pro.

1.2 Dissertation Overview

This chapter has introduced some of the unique challenges present in comprehending large low-level systems written in assembly language, such as difficult decomposition of control flow graphs, and motivated the need for tool support beyond what is now considered state-of-art for assembly language.

In response, we have created AVA (Assembly Visualization and Analysis), a program comprehension framework that is designed to be applied to specific challenges for multiple assembly languages. The particular languages we support are HLASM and x86, which are used by our stakeholders. Our initial assumption for AVA was that tools could be built to encompass the universe of all assembly languages. This assumption was made previous to requirements elicitation, and detailed understanding of the technical differences between the languages.

In order to build AVA, our first and second research questions were:

- *What are the requirements currently not being met in the comprehension of assembly code within two unique groups: mainframe developers and malware analysts?*

and

- *What are the similarities and differences in the requirements?*

Phase II In order to answer the above questions, we conducted interviews, issued an exploratory survey, as well as elicited requirements within a team at each group. Our results show that while the same categories of requirements exist, the majority of requirements within those categories are specific to the domain in which they were elicited. However, the requirements do show that tool support is lacking for each group, and the specific ways in which it can be improved. This brought us to our third research question:

- *Can program comprehension tools for high-level languages be retrofitted to apply to low-level languages?*

Phase III, Part I As mentioned previously, we have created the Assembly Visualization and Analysis (AVA) tool framework, which comprises the re-purposing of tools originally designed for the comprehension of control flow and structure within

high-level languages. While we were successfully able to retrofit these tools, the fourth and final research question we ask is:

- *Are tools in our proof of concept framework effective at supporting the requirements of both groups?*

Phase III, Part II In Part I, we discuss the design and implementation of three proof of concept tools in our AVA framework. While one of these tools, called Tracks, is found to be effective for both groups of stakeholders, we find that the other two tools within this framework—LegaSee and REwind—are not. Though initial surveys and requirements elicitation foreshadowed this result, we further assess through case studies using each of the tools in mainframe and malware contexts.

Following these questions, the claim of this dissertation is that: *While program comprehension tools can be effectively applied to low-level programming languages, such as assembly language, they cannot be universally applied due to their specialized use in industrial software groups, compounded by fundamental construct differences.*

We demonstrate our claim by investigating two different assembly language dialects in use within two specialized groups, and show that:

1. There exists a minimal intersection of requirements between these two highly specialized industrial software groups;
2. High-level program comprehension tools can be retrofitted to work with low-level language constructs;
3. Fundamental differences in both languages and groups necessitate a disjoint set of tools that do not benefit from consolidation into a unified framework.

To affirm our first point, we provide the results of surveys and requirements elicitation in Chapters 3 and 4 respectively, comparing and reasoning about the differences between groups in Chapter 5. In Chapter 6, we discuss the design and implementation of the Assembly Visualization and Analysis (AVA) tool framework built in order to illustrate our second point. The third and final point is demonstrated in Chapter 7, which provides case studies using our proof of concept tools, showing that while specific tools can be built for universal use, others are specialized to the architecture and/or stakeholders involved. Chapter 7 concludes with further explanation of our findings, drawing on results from Chapter 5, as well as technical issues brought to light in Chapter 6. The dissertation outline is shown in Figure 1.3.

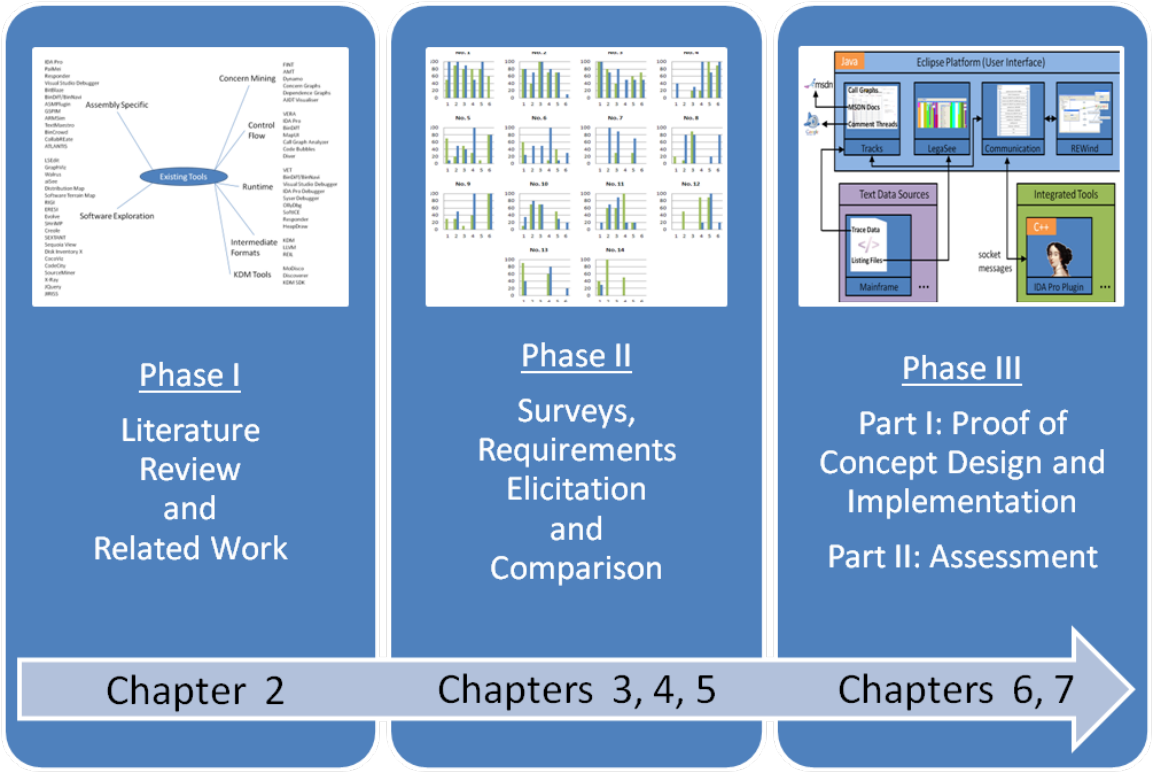

Figure 1.3: Dissertation Overview.

1.2.1 Agenda

This dissertation is outlined as follows:

- **Chapter 2** provides a survey of related work in both requirements elicitation and tool support. We end with an introduction of technologies that are used in the implementation within this dissertation.
- **Chapter 3** provides the results of preliminary interviews within the mainframe stakeholder domain, as well as results of a large exploratory survey that was conducted with each group of stakeholders. We finish the chapter by providing a more in-depth survey of collaboration and documentation within the malware analyst group.
- **Chapter 4** provides the research method and results of requirements elicitation performed with each group of stakeholders.
- **Chapter 5** provides a comparison between the mainframe developer group, and malware analyst group. We further discuss why both groups have unique

needs based on their particular area of expertise. This chapter demonstrates how the same set of tools cannot accommodate the needs of both groups.

- **Chapter 6** introduces the tool support built to address needs discussed in previous chapters. We discuss the implementation of these tools, as well as work completed by colleagues within the scope of this project.

- **Chapter 7** provides an assessment of this dissertation. We analyze the requirements elicitation process that we followed in Chapter 4. Following this, we consider the assessment first in terms of the mainframe developer domain, followed by the malware analyst domain.

- **Chapter 8** concludes with a summary of future work, and our findings. This future work includes further studies in this context, as well as further feature development possible for the tools we have created thus far.

- **Appendices** There are six appendices included within this dissertation. The first appendix is a summary reference of results from the exploratory survey. The second appendix provides the script for the group session used for requirements elicitation. The third appendix lists issues observed during activity-based elicitation within the mainframe context. The fourth appendix describes how to download and install the AVA tool framework, both for use and for development. The fifth appendix explains how to run the Mariposa botnet within IDA Pro, which is useful in seeing how our control flow tool can create sequence diagrams for malware. The sixth and final appendix provides our ethics approval for the study.

Chapter 2

Related Work

This chapter provides the background information necessary in understanding Chapter 4 on Requirements Elicitation, as well as Chapter 6 on Design and Implementation. We also provide a survey of related tool support in this chapter. While the majority of these surveyed tools are not discussed again in this dissertation, they provide an important landscape from which prototypes beyond the scope of our work could be derived.

2.1 Social Psychology Background

This section outlines the social psychology theory and background necessary to understand our requirements elicitation process.

2.1.1 Normative Manipulation

From a psychological perspective, requirements elicitation can be seen as an information sampling task: in order to be successful, groups of people have to get together to share relevant information. Information sharing within a group, however, can be moderated by group effects such as *production blocking* and *groupthink* [7, 8]. Production blocking occurs when one person blocks others during a group session, for example when only one person can speak at a time. Similarly, groupthink occurs when a group tends towards harmony and rejects any input that is contrary to their views.

Group norms are defined as groups developing personalities or identities of their own which override individual personalities in the group [9]. This can be a problem

for requirements elicitation, for example, when people are reluctant to express ideas during a brainstorming session because they may be judged by the group. Postmes et al. [10] discuss how priming was used to discourage groupthink.

Group norms can be manipulated (through *normative manipulation*) so that group members adopt a norm that is predicted to be beneficial for the task at hand. The only previous work we are aware of that explores using these group norms for requirements elicitation in software is provided by Teh et al. [11]. Their study used two group norm conditions: *consensus* and *critical*. A consensus group norm leads to an establishment of groupthink, whereas a critical group norm prevents people from converging too quickly. They show that with two similar groups, the critically primed group produced, on average, more unique and correct requirements for an elicitation task.

The requirements elicitation process used in this dissertation adopted the social psychology technique of normative manipulation to promote the sharing of requirements by participants. However, the type of normative manipulation that can be applied is dependent on the profiles of the participant groups. These profiles are based on two social psychology measures, the *Need-for-Closure* [12] and *Individualism-Collectivism* [13] scales.

2.1.2 Need-for-Closure (NFC) Scale

The NFC scale was originally proposed by Kruglanski [12], and is defined as the desire for *"an answer on a given topic, any answer... compared to confusion and ambiguity"*. High NFC individuals tend to be closed-minded and do not appreciate having their opinions challenged [14].

The NFC predicts *epistemic motivation* or willingness to expend effort to achieve a thorough and rich understanding of the world [15]. Epistemic motivation is associated with trait-based openness to experience and may indicate how willing participants are to adopt new tools during future studies. It may also correlate with how our requirements elicitation process was received.

2.1.3 Individualism-Collectivism (INDCOL) Scale

The INDCOL scale is a measure of how independent individuals are of one another, or, to what degree groups are bound and mutually obligated to one another [16]. We use this measure to determine the type of normative manipulation to apply.

It was shown by Goncalo and Staw [17], that individualistic groups generate significantly more ideas when they are instructed (primed) to be creative, whereas collectivist groups generate somewhat more ideas when they are told to be practical. Additionally, individualistic groups that are instructed to be creative generate a higher percentage of creative ideas, whereas there is no significant difference between collectivist groups instructed to be creative rather than practical. Goncalo and Staw defined creativity by ideas that were both novel and useful. These studies were done by not only priming creativity/practicality but by first priming the individualistic or collectivist mindset with a series of questions to make participants think either about themselves, or about their group.

2.1.4 Group Decision-Making Techniques

There were three possible decision-making group techniques that we could employ: Interacting Group, Delphi Group and Nominal Group [18].

The Interacting Group is the most common form where team members talk amongst themselves and come to a decision through arguments and agreements. The main advantage is that new ideas can be generated, and understanding improved. The major disadvantage is that political process can play a large role.

The Delphi Group is used to develop group agreement about the relative importance of issues [19]. The process is similar to the nominal technique in that there exists the generation of ideas based on a question, and ranking of these ideas. But the process differs in that ranked lists of the original ideas are sent out repeatedly to each expert for re-ranking, until agreement by calculated mean rank is achieved. This means that the process may be timely and participants do not need to be co-located, an advantage in some circumstances.

The Nominal Group is a structured process where ideas are generated based on a question, but the issues are discussed and ranked within the group session. We selected this technique for this dissertation. It was chosen over the Interacting Group since we wanted to avoid issues of pre-existing hierarchy within the team, as well as issues such as production blocking and groupthink. The Delphi Group was also not chosen since it is the most time-consuming of all methods, and is best when participants are not co-located.

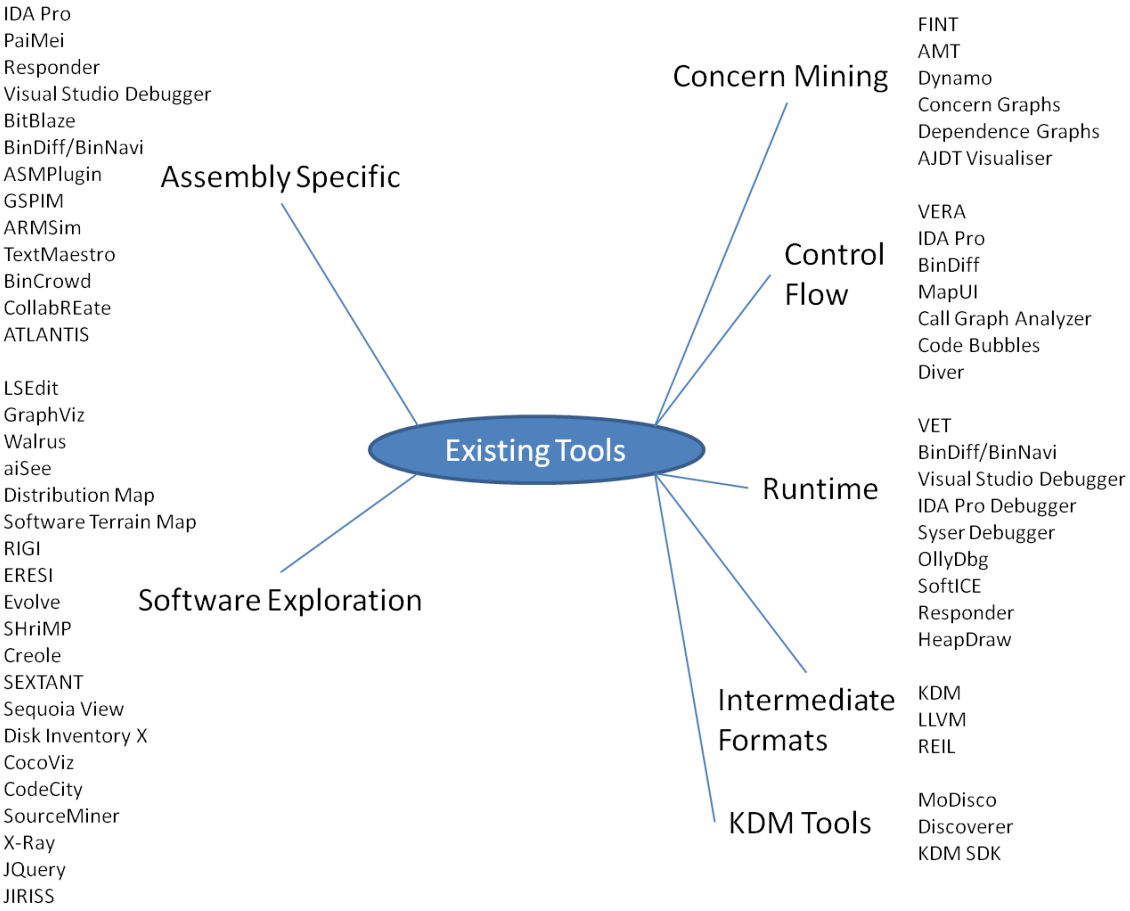

Figure 2.1: Overview of Related Work.

2.2 Related Work in Tool Support

This section summarizes areas of tool support that we believe are important within the scope of this dissertation. Figure 2.1 shows these areas as well as the tools we have found within. Some tools may appear in more than one area.

We first discuss assembly specific tools to see what exists in the problem space, followed by software exploration tools which are abundant, but only a handful are geared towards assembly. Concern mining has not been applied to low-level languages and could be an interesting avenue of research. For example, Figure 1.1 may benefit from separation of control flows into chunks based on concerns, providing a better way to understand a system than trying to figure out intertwined `LOAD`s and `GOTO`s. Control flow tools are of great interest, as we will see later in this dissertation, and some tools already exist to aid with low-level languages. Interestingly enough, many

runtime tools exist for assembly, yet developers are still asking for different features such as register usage and propagation, as well as being able to step back and forth through program execution. Finally in the interest of creating language agnostic tool support, we look at potential intermediate formats, as well as tools that exist for one of those formats—Knowledge Discovery Metamodel (KDM) [20].

2.2.1 Assembly Specific Tools

This set of tools is a collection of what could be found for assembly language and are not necessarily visual tools, but some of their features may be useful in the future.

Some tools that are currently employed in industry for assembly include IDA Pro [5], a disassembler and debugger, PaiMei [21], which is a reverse engineering framework, Responder [22], a runtime and memory analysis tool, and Visual Studio's debugger. BitBlaze [23], is a binary analysis platform to analyze, understand, and develop defenses against malicious code. Zynamics' BinDiff and BinNavi [24] are also tools for comparing, and analyzing, disassembled code respectively.

ASMPlugin [25] is an assembler plugin for Eclipse that includes an assembler editor (with syntax highlighting) and includes a linker and debugger. GSPIM [26] is used for visualization of low-level MIPS Assembly programming and simulation. There has also been some work at University of Victoria with ARM simulation for education purposes [27]. Finally, TextMaestro [28] converts assembly language to its corresponding C code.

There are two tools, that we are aware of, that try to address collaboration within reverse engineering tools. These are BinCrowd by Zynamics [24] and CollabREate [29, 30]. BinCrowd provides a database that can be used to share disassembly information with team members. This information can then be shared via a web interface or through an IDA Pro plugin. It works by allowing an analyst to document code and then upload these comments to the database. BinCrowd then uses its BinDiff format to find related functions, and files, as well as comparing files and displaying statistics about them. These comments are inlined within the code using IDA Pro's commenting capability. Though reasonable as a means of documentation for a single developer, this form of commenting may not be conducive to discussion and collaborative documentation. Figure 2.2 shows the importing of comments to functions within IDA Pro using the BinCrowd plugin. In this case, two functions are identified as having a high match quality.

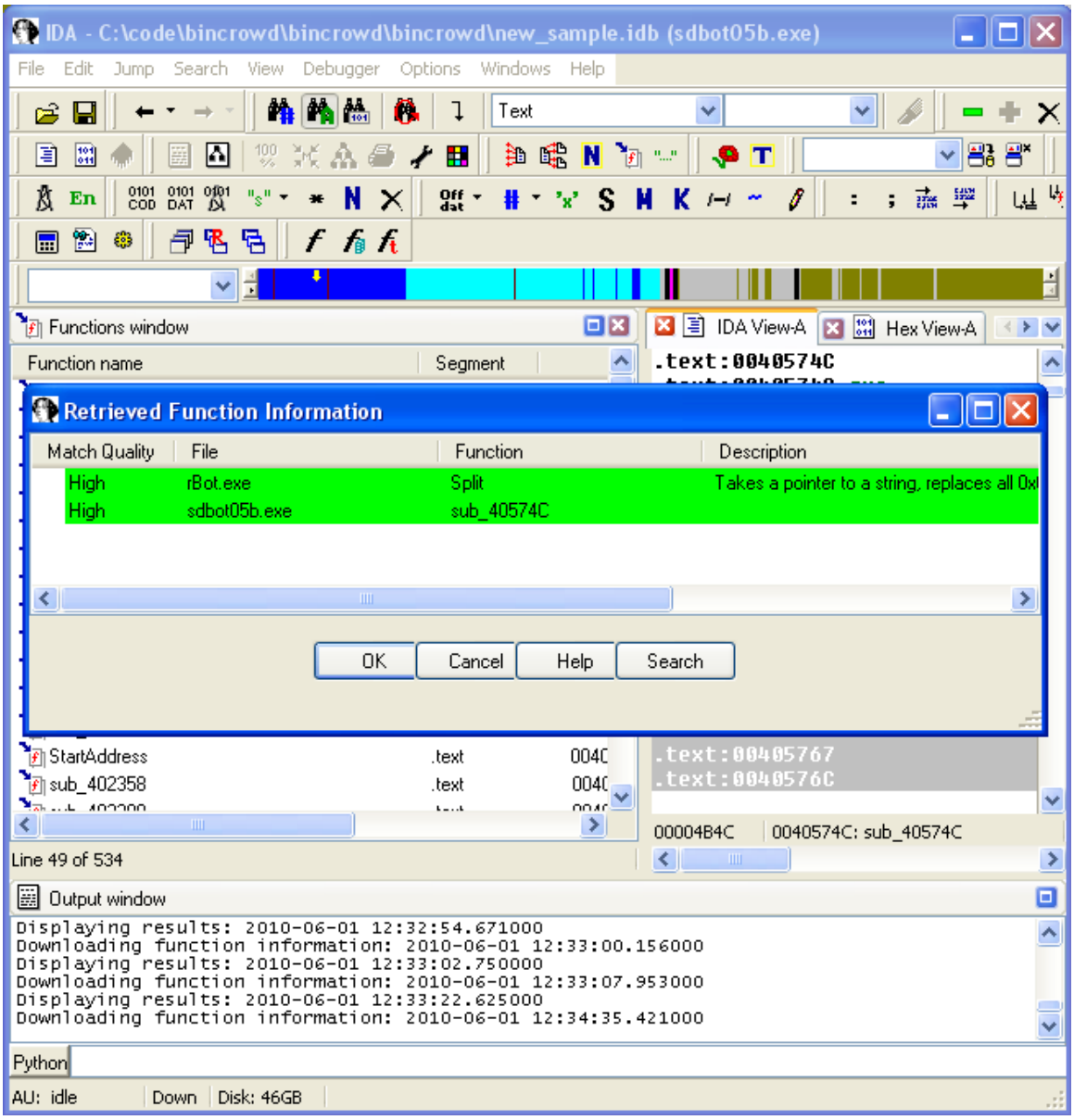

Figure 2.2: BinCrowd in IDA Pro.

CollabREate works somewhat similarly to BinCrowd in that it uses an IDA Pro plugin and pushes changes to IDA Pro database (IDB) files to a server. Any user that is subscribed to the same CollabREate project will receive these updates, and therefore remain synchronized. These IDB updates include comment changes, but also quite a few more including adding or deleting functions, enums and structs. Again, these comments surround functions, but may not be sufficient to support collaboration.

There are other tools that have been created by colleagues during the course of this project, some of which are discussed at the end of Chapter 6. However, one tool that is not discussed in Chapter 6 is ATLANTIS (Assembly Trace Analysis Environment) [31]. While related to our project, this is a separate project undertaken by the CHISEL group at the University of Victoria, and DRDC Valcartier. ATLANTIS aids malware analysts in identifying exploits in software by using execution traces, rather than the original source code. Figure 2.3 shows an example of this environment.

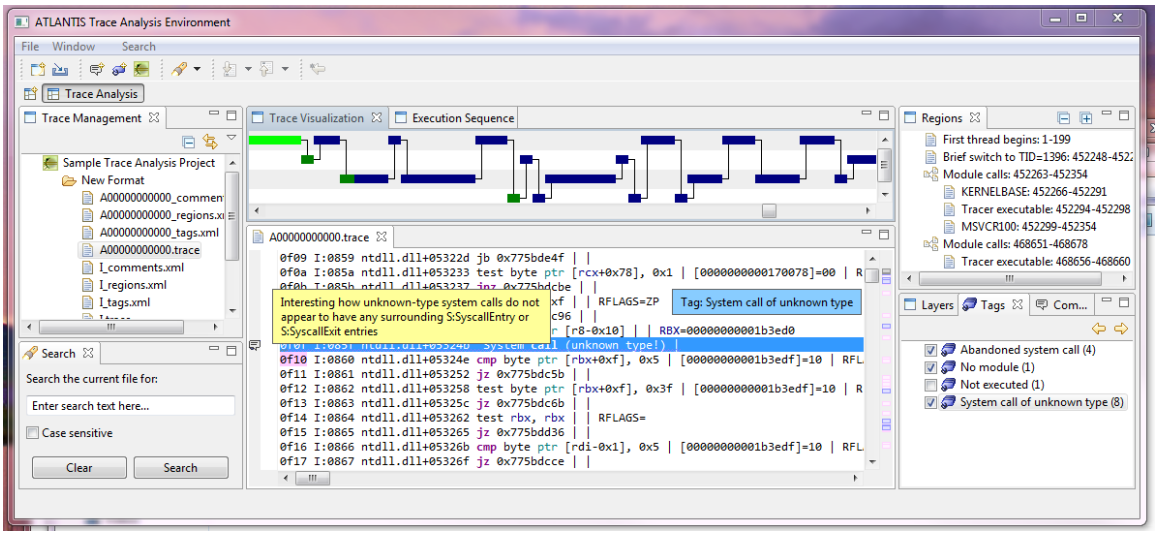

Figure 2.3: ATLANTIS (Assembly Trace Analysis Environment).

2.2.2 Software Exploration Tools

Software exploration is an active area of research which provides tools that allow a user to explore a software system visually. This includes call graphs [32], graph-based tools such as LSEdit [33], GraphViz [34], Walrus [35], aiSee [36] and distribution [37] and terrain maps [38], reverse engineering tools like PaiMei [21], Rigi [39] and ERESI [40],

and visualization and exploration tools, for example, Evolve [41], SHriMP [42] and SEXTANT[43]. There are also many tools in the area of tree maps such as Sequoia [44] and Disk Inventory [45]. Others use a city analogy to represent architecture such as CocoViz [46] and CodeCity [47], and many are built as Eclipse plugins, including SourceMiner [48], X-Ray [49], JQuery [50] and JIRISS [51].

2.2.3 Concern Mining Tools

These tools locate feature implementations, or concerns, within the code and are able to automatically extract them, and potentially view them in some way. These tools include FINT [52], the Aspect Mining Tool (AMT) [53] and others [32, 54, 55, 56]. Additionally, Robillard et al. [57] present a technique using concern graphs, which are abstract models that describe which parts of the source code are relevant to different concerns. We believe these tools may be interesting to look at in order to abstract features from assembly code, which could aid in control flow decomposition as well as in other areas.

2.2.4 Control Flow Tools

While it is relatively easy to understand a few lines of assembly, the problem is much more difficult when trying to understand thousands of lines of code. One quickly gets lost, especially if the code is obfuscated. There has been little work in the area of control flow visualization for malware analysis and assembly in general. Most notably, VERA (Visualization of Executables for Reversing and Analysis) [58], presents a graph that uses basic blocks as nodes to support dynamic analysis. In most tools, the visualization is limited to a static function call graph, such as in IDA Pro and Bin Navi [5, 24]. Other tools are limited to text interfaces with very little access to modern development environments' UI features.

VERA provides a high-level dynamic view of basic blocks, loops and color coding to describe where code is located. It also provides navigational links to IDA Pro. While it allows users to quickly pinpoint areas of interest, it is not very useful after this point. The user must return to IDA Pro to understand the finer details.

One effort towards static control flow information for assembly is a tool called MapUI. DeLine [38] and others came up with the idea to represent source code with a software terrain map to use the brain's spatial memory. DeLine came to the conclusion that there are better representations for source code (e.g. Code Thumbnails [59]).

However, the team at DRDC decided to try software terrain maps with assembly code since many of the other representations cannot be applied. Figure 2.4 shows a software terrain map from the MapUI prototype of the Zeus builder [60, 61]. The Zeus crimeware toolkit is a set of programs which are designed to set up a botnet over a high-scale networked infrastructure.

Each function is represented as a country. The size of the country depends on the number of lines of assembly in that function. The relationship of countries is calculated by the number of static calls to/from each function. The functions with greatest affinity are laid out first. Continents or islands are created when there are no links to other functions (e.g. exception handling, dead code). Then colors can be applied and layers can be created according to different concepts. For example, in Figure 2.4, red represents functions related to building a customized bot and green represents functions related to removing the spyware. This functionality is provided by the Zeus builder in case the hacker accidentally infects himself.

For languages other than assembly, Bohnet and Döllner [32] show the value of visually exploring call graphs to find features in large C/C++ systems (over 1 million lines of code). Their approach is a control flow graph that combines dynamic and static analysis techniques.

Other control flow tools exist for higher-level languages such as Code Bubbles [62], an IDE for Java to create bubbles containing methods that are linked when the user selects a static call in the source code. They also provide bubbles for notes and status flags for easy documentation. However, while many of the features of Code Bubbles are useful, its focus is more on the code within the bubbles. There is no view that contains just the calls. It also does not focus on extremely large traces which we must contend with in assembly code.

Finally, Diver [63] is an open-source, extensible Eclipse-based framework for creating sequence diagrams. Diver contains an example implementation which provides sequence diagrams for Java. We discuss Diver later in Section 2.3.1.

2.2.5 Runtime Tools

This set of tools is helpful for a developer to discover information during the execution of their system. The tools we look at include the Visualization Execution Tool or VET [64], which helps programmers manage the complexity of execution traces, and also other tools for debugging such as Bin Diff and Bin Navi [24], the Visual Studio

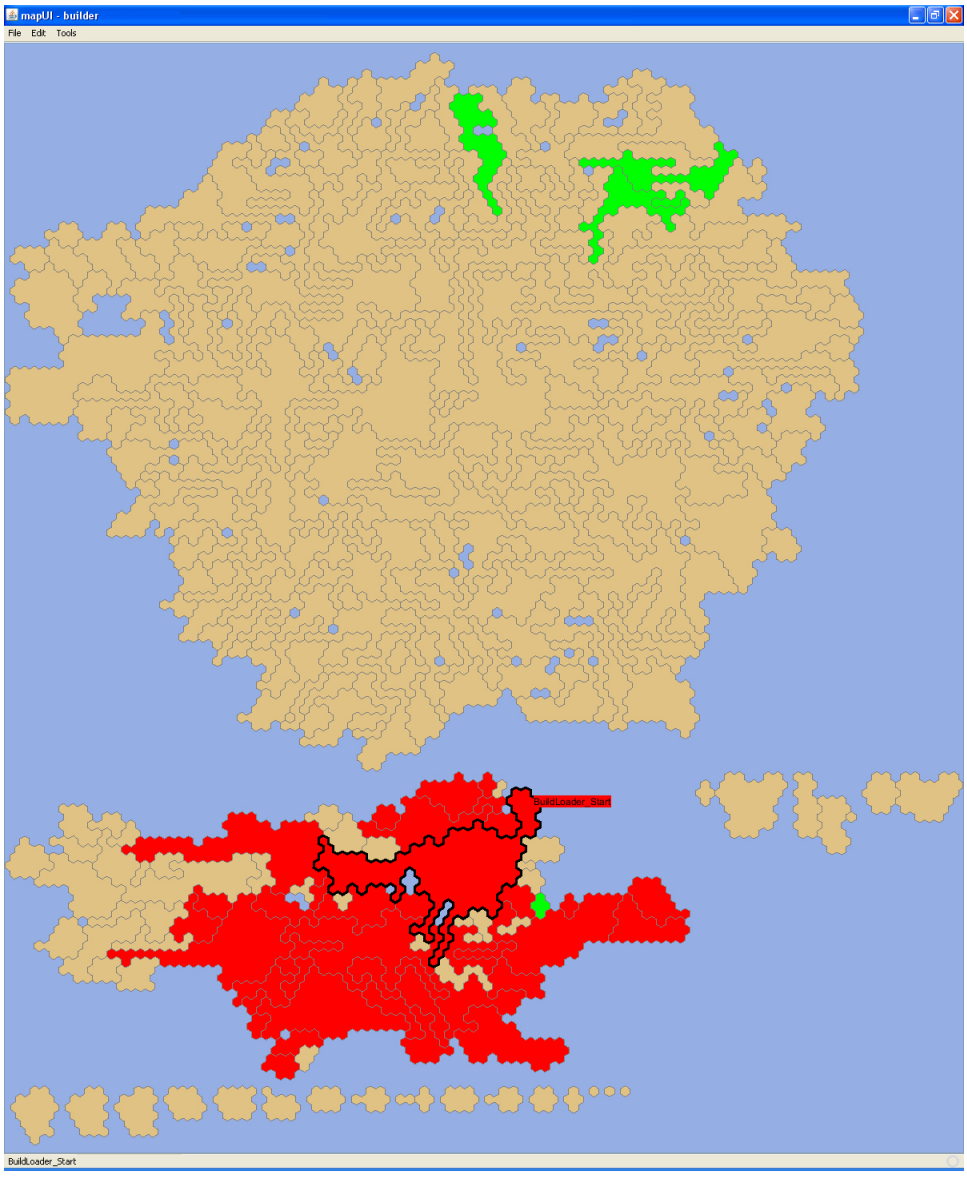

Figure 2.4: MapUI and Zeus' 375 Functions (23,320 Lines).

debugger, the IDA Pro debugger [5], the Syser debugger [65], OllyDbg [66], and SoftICE [67], and memory analysis tools like Responder [22] and HeapDraw [68]. We believe this class of tools is important in helping developers identify memory leaks, buffer overflows, causes of segmentation faults, as well as understand how registers and their values are propagated throughout the system.

2.2.6 Intermediate Common Formats

One of the objectives of the initial project proposal was to establish if a common intermediate representation for a large set of assembly languages would be possible. In particular, what is necessary to support abstractions required for comprehension tasks. While not a contribution of this dissertation, other colleagues have investigated potential existing representations such as the Knowledge Discovery Metamodel (KDM) [20], the Low Level Virtual Machine (LLVM) object code representation [69] that uses simple RISC-like instructions, and leaner experimental candidates such as Zynamics' Reverse Engineering Intermediate Language (REIL) [70].

Ultimately, no existing intermediate language (IL) could support our goals. That is, to be the one IL for multiple dialects, as well as provide the specific information necessary for our intended comprehension tools [71]. Due to these limitations, we have created our own data format discussed in the beginning of Chapter 6.

As an example of challenges faced with existing metamodels, let us look more closely at KDM. Unfortunately, KDM produces enormous XML files to describe only small amounts of code. For example, a KDM file for a HelloWorld.c program is 42 lines, shown in Figure 2.5. Therefore, it was not a viable option for this project. Nevertheless, we investigated possible tools for KDM in case they could be applicable.

The most interesting tool for KDM is MoDisco [72], which can create UML diagrams from KDM code. It also includes a tool called the Discoverer, which discovers a full abstract syntax tree for Java, and builds models from it. Additionally, there is the Knowledge Discovery Metamodel SDK [20], an Eclipse plugin which provides a set of tools for working with KDM.

2.3 Implementation Technologies

This section discusses the technologies we utilized to create the prototype tools in our AVA framework. Our base technology for building these tools was Eclipse. This

```
<?xml version=''1.0'' encoding=''UTF-8''?>
<kdm:Segment xmi:version=''2.1''
xmlns:xmi=''http://www.omg.org/XMI''
xmlns:action=''http://kdm.omg.org/action''
xmlns:code=''http://kdm.omg.org/code''
xmlns:kdm=''http://kdm.omg.org/kdm''
xmlns:source=''http://kdm.omg.org/source''
name=''HelloWorld Example''>
  <model xmi:id=''id.0'' xmi:type=''code:CodeModel'' name=''HelloWorld''>
    <codeElement xmi:id=''id.1'' xmi:type=''code:CompilationUnit'' name=''hello.c''>
      <codeElement xmi:id=''id.2'' xmi:type=''code:CallableUnit'' name=''main'' type=''id.5'' kind=''regular''>
        <source xmi:id=''id.3'' language=''C'' snippet=''int main(int argc, char* argv[]) {}''/>
        <entryFlow xmi:id=''id.4'' to=''id.12'' from=''id.2''/>
        <codeElement xmi:id=''id.5'' xmi:type=''code:Signature'' name=''main''>
          <source xmi:id=''id.6'' snippet=''int main(int argc, char * argv[]);''/>
          <parameterUnit xmi:id=''id.7'' name=''argc'' type=''id.25'' pos=''1''/>
          <parameterUnit xmi:id=''id.8'' name=''argv'' type=''id.9'' pos=''1''>
            <codeElement xmi:id=''id.9'' xmi:type=''code:ArrayType''>
              <itemUnit xmi:id=''id.10'' type=''id.19''/>
            </codeElement>
          </parameterUnit>
          <parameterUnit xmi:id=''id.11'' type=''id.25'' kind=''return''/>
        </codeElement>
        <codeElement xmi:id=''id.12'' xmi:type=''action:ActionElement'' name=''a1'' kind='''Call''>
          <source xmi:id=''id.13'' language=''C'' snippet=''printf(&quot;Hello, World!\n&quot;);''/>
          <codeElement xmi:id=''id.14'' xmi:type=''code:Value'' name=''&quot;Hello, World!\n&quot;'' type=''id.19''/>
          <actionRelation xmi:id=''id.15'' xmi:type=''action:Reads'' to=''id.14'' from=''id.12''/>
          <actionRelation xmi:id=''id.16'' xmi:type=''action:Calls'' to=''id.20'' from=''id.12''/>
          <actionRelation xmi:id=''id.17'' xmi:type=''action:CompliesTo'' to=''id.20'' from=''id.12''/>
        </codeElement>
      </codeElement>
    </codeElement>
    <codeElement xmi:id=''id.18'' xmi:type=''code:LanguageUnit''>
      <codeElement xmi:id=''id.19'' xmi:type=''code:StringType'' name=''char *''/>
      <codeElement xmi:id=''id.20'' xmi:type=''code:CallableUnit'' name=''printf'' type=''id.21''>
        <codeElement xmi:id=''id.21'' xmi:type=''code:Signature'' name=''printf''>
          <parameterUnit xmi:id=''id.22'' name='''' type=''id.25'' kind=''return'' pos=''0''/>
          <parameterUnit xmi:id=''id.23'' name=''format'' type=''id.19'' pos=''1''/>
          <parameterUnit xmi:id=''id.24'' name=''arguments'' kind=''variadic'' pos=''2''/>
        </codeElement>
      </codeElement>
      <codeElement xmi:id=''id.25'' xmi:type=''code:IntegerType'' name=''int''/>
    </codeElement>
  </model>
  <model xmi:id=''id.26'' xmi:type=''source:InventoryModel'' name=''HelloWorld''>
    <inventoryElement xmi:id=''id.27'' xmi:type=''source:SourceFile'' name=''hello.c'' language=''C''/>
  </model>
</kdm:Segment>
```

Figure 2.5: Example KDM for Hello World Program Written in C.

choice is discussed further in Chapter 6.

First we discuss the sequence diagram tool used to create control flow graphs. Second we introduce a graphical editor and graph-creation tool, and finally an Eclipse plugin for visualizing system constructs at a high-level.

2.3.1 Diver: The Sequence Explorer for Eclipse

One of the more difficult challenges in understanding assembly code is following control flow. This is due to the inherent, unstructured nature of the code. To create Tracks, we extended Diver, an open-source and extensible sequence diagram tool built using the Eclipse framework [73]. The design of Diver has two primary goals: model independence and interactivity/navigability.

Much work in industry and in research has been spent implementing multiple instantiations of very similar visualizations for program control flow. Therefore, a need was seen for a reusable, interactive sequence diagram viewer in order to eliminate duplicate work.

Model-independence means that the viewer is not tied to any particular model or data format in its back-end. The viewer has been employed to visualize program control flow from various sources. Such sources include control flow of assembly language instructions (in this research), dynamic traces from instrumented Java programs [74], and call structures of static Java source code. This has been accomplished by using a framework compatible with the Eclipse JFace [75] viewer framework. This means that implementors must write some Java code in order to realize their application, but they are also abstracted far away from the details about how to draw the lines, boxes, and labels necessary for displaying the view.

The second goal of interactivity and navigability was inspired by the fact that sequence diagrams can quickly become very large and extremely complex. Diver has integrated features to help overcome this problem. A short listing of the features includes: animated layout, highlighting of selected elements and related sub-calls, grouping of related calls (such as loops), hiding/collapsing of call trees and package or module structures, customisable colors and labels for visual elements such as activation boxes and messages, keyboard navigation through components, and the ability to reset (focus) the sequence diagram on different parts of the call structure. These features have been extensively studied and evaluated [74, 76]. Figure 2.6 shows an example of Diver visualizing a sequence diagram of a Java program.

2.3.2 GEF: Graphical Editing Framework

The Graphical Editing Framework (GEF) [77] bundles three components that are used to create graphical editors within Eclipse. The GEF component of particular interest is Zest, which is a visualization toolkit used to create graphical views within Eclipse. The Zest project contains within itself a graph layout package that includes various layout algorithms. Apart from the use of Zest in Diver, we also use Zest in the creation of the state diagram tool that we see in Chapter 6 on implementation.

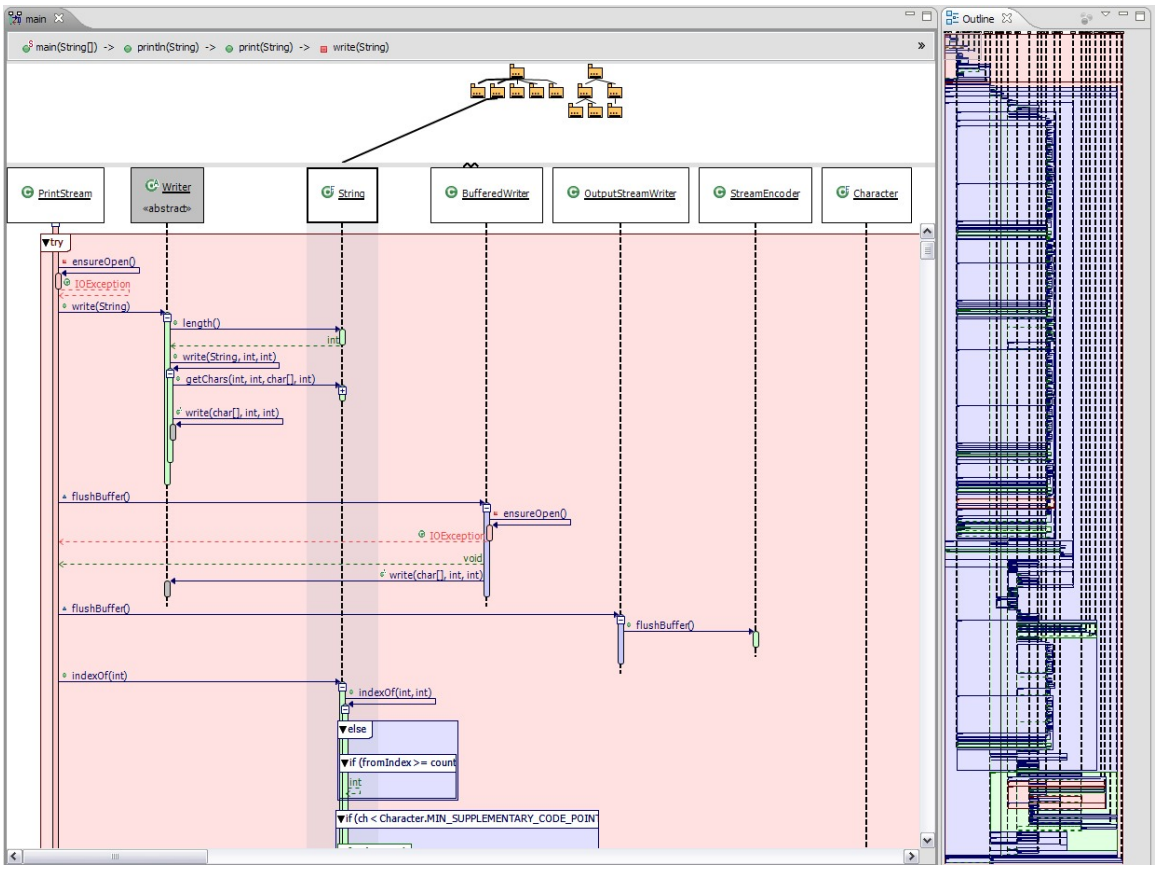

Figure 2.6: Diver used with a Large Java Project.

2.3.3 The AJDT Visualiser

In order to visualize certain aspects of assembly code at a high-level, we needed some sort of tree map. The Visualiser [78] is not a tree map, but somewhat similar with its use of colored stripes and blocks. It is also freely available and easily extended, which is why it was selected as a first step towards a scalable tool for this purpose.

The Visualiser is an extensible Eclipse plugin, originally part of AspectJ Development Tools (AJDT), that can be used to visualize anything that lends itself to a 'bars and stripes' style representation. It began as the Aspect Visualiser, which was used to visualize how aspects were affecting classes in a project. It did so by showing each class as a bar, with its length corresponding to its length in lines of code. Each aspect was then color-coded and drawn as a stripe based on its location and number of affected lines of code (or lines of code in the aspect itself). The Visualiser provides extension points and there are a few publicly available providers, including those for Google searches and CVS history. We have also used it before in the context of patch

tool support for systems code [79].

Figure 2.7 shows an example of an AspectJ project and how aspects have crosscut files within the system. One can also switch to a package view, which shows how packages are affected. Navigation is supported by double-clicking stripes, and the colors used are customizable. Each different menu item can be toggled in order to hide them from the visualization.

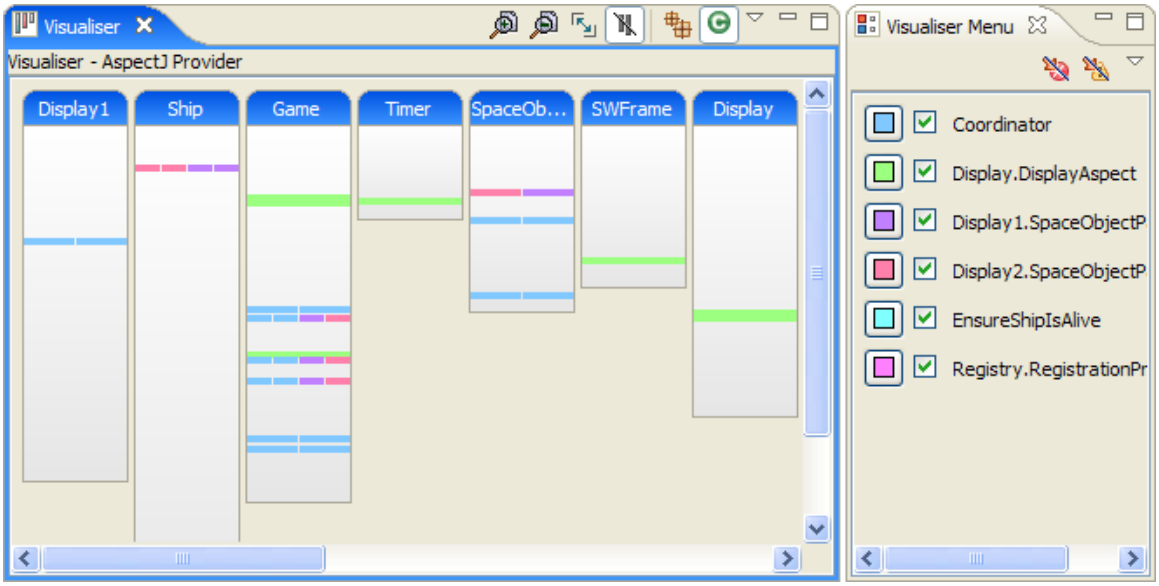

Figure 2.7: The AJDT Visualiser.

2.4 Chapter Summary

This chapter provided the groundwork in the social psychology methods and measures that are used in our requirements elicitation. We have also provided a survey of multiple tools in the different areas that we believe are necessary in the context of comprehension support for assembly language. These areas include: Assembly-Specific, Software Exploration, Concern Mining, Control Flow, Runtime, and Intermediate Formats. While we believe a combination of features from these tools are needed to effectively assist developers in understanding and maintaining low-level software, there is no existing framework of tools for the comprehension of assembly language programs specifically. Finally, we have introduced the background of technologies that will be seen in Chapter 6 on Design and Implementation.

Chapter 3

Exploratory Interviews and Surveys

This chapter provides the results of three separate exploratory studies. Our first study involved four interviews with engineers working with mainframe code. The second study was a large exploratory survey which encompassed experience, current work, browsing and navigation, control flow, and debugging. The final study takes a closer look at the issues of malware collaboration and documentation. Ethics approval for this study is shown in Appendix F. These issues are investigated further since they emerged from the malware analysts in the exploratory survey.

3.1 Interviews with Mainframe Developers

In March 2009, we visited the CA Prague Technology Center (PTC). CA PTC is known primarily for its mainframe and distributed computing applications and solutions used by businesses. Here we met with five developers from various backgrounds. They were able to freely discuss their issues as well as what they believed was fundamental for their understanding. Their stories are summarized below.

3.1.1 First Engineer

Rob[1] was an experienced assembly developer who was working with an extremely large module. Rob spent some time showing us his mainframe development environment, and how he often worked with the code. He also told us which the most important tools that he wanted were. These included connections between modules, analyzing

[1]The names used in this study are fictitious for the purposes of preserving individual privacy.

subroutines (HLASM does not have functions, only coding conventions), support for dummy sections (DSECTs) and control sections (CSECTs), as well as register usage.

In order to find DSECTs, Rob was using text search. There was no way to see where they were defined or used at a high level, and no navigation to them.

3.1.2 Second and Third Engineer

The second interview we had was with two engineers, David and Joe, who both worked on a database written in assembly. When bugs occurred in the system, usually it was due to an instruction modifying a DSECT. When this occurred, they had to use a cross-reference tool to find everything that modified the DSECT, and it had some shortcomings.

David and Joe wanted something similar to a debugging tool, which would allow them to jump through modules to follow data flow by using a log file. Navigation would allow them to move back and forth with their selections and also back and forth with how the program ran, with the addition of breadcrumbs to show this. They also needed to know the values in each register. Additionally, they wanted to have architectural diagrams that developers could collaborate on. Figure 3.1 shows their mockup of this tool.

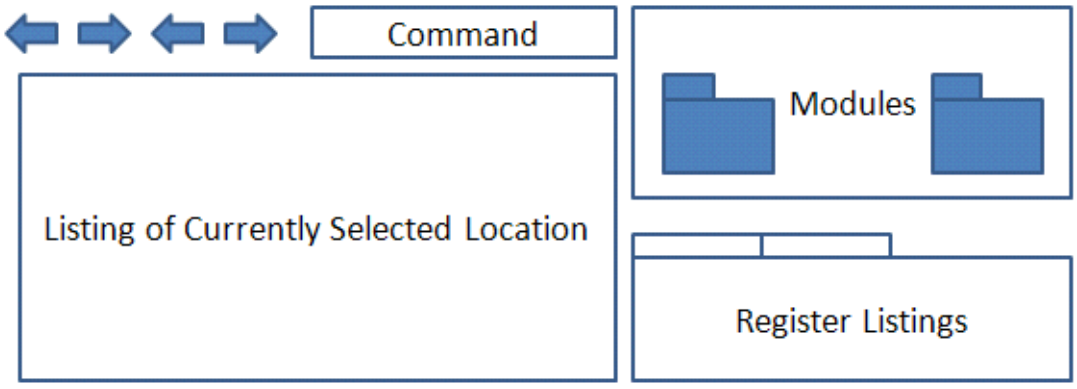

Figure 3.1: Mockup UI Design for Mainframe Debugging Tool.

3.1.3 Fourth Engineer

Alex had created an Eclipse plugin that provides syntax highlighting for HLASM assembly, as well as the ability to upload/download files from the mainframe and

execute them. It could also view log files as well as the outline of the assembly code (macros, DSECT, CSECT).

Future work that Alex wanted for this tool included code completion and syntax support in the editor, so the code would not have to run on the mainframe before syntax errors were shown. Also being able to search and graph references for where symbols are used (or defined).

3.1.4 Fifth Engineer

Bill had been given the task of trying to understand a huge assembly system called *reportbroker*, which has 493 modules. He had written a tool that separates data from the listing into two different files, one for the source code and one for the usage of each label, varname, etc. by module. This was because it was important for him to know which modules to look at when he was dealing with a specific label or variable name.

3.1.5 Summary of Interviews

In looking at the results of these interviews, there seem to be three separate areas of concern:

1. Development Tools
2. Debugging/Runtime Tools
3. Visualization/Comprehension Tools

The development tools would include syntax highlighting/checking, code completion and being able to search for references, basically common IDE support. Debugging tools would allow viewing values of registers at runtime, or from a log file, and stepping through the system. Finally, visualization and comprehension tools would include tools such as those for control flow, data flow, references, and architectural diagrams.

Table 3.1 shows a summary of the results from the interviews with developers. To briefly summarize this Table, the first and last developer were both experienced with assembly but still had difficulties. These included connections between modules, identifying subroutines, understanding and locating the use of DSECTs and CSECTs, seeing how register usage is propagated throughout the system, and understanding

where variables and labels are used throughout modules in the system. The second team of developers were maintaining a mainframe database and had many issues debugging their system, since most errors were caused by a runtime modification to a dummy section. The third developer had already created an Eclipse plugin for his own use that included syntax highlighting and integration for syncing files with the mainframe, but still lacked many features such as syntax checking, code completion and reference lists and graphs.

Interview	**Issues**
1. Assembly Developer	Connections between modules, identification of subroutines, DSECT and CSECT support, register support.
2. Database Developers	DSECT modification, debugging tools, data flow.
3. Eclipse Plugin Creator	Syntax highlighting/checking, integration with mainframe, code completion, reference lists and graphs.
4. Assembly Developer	Separate listings into: - source code - modules using each label/variable name

Table 3.1: Summary of Interviews with the Mainframe Group.

3.2 Exploratory Survey

This section discusses the results of a large exploratory survey that was distributed to two separate groups. The first group included our stakeholder contacts within the mainframe systems group, as well as those who responded from a public invitation made on the IBM Mainframe Assembler mailing list. The second group included those that work with security issues, and included our stakeholders in the malware group, as well as their clients and trainees, and students within the computer security laboratory at Concordia University. We had 25 participants from the mainframe group and 15 from the malware group.

This exploratory survey looked at seven factors: respondent experience and preferences with programming in general, familiarity with assembly language, their current toolset for assembly, browsing and navigation concerns, features of debugging, control flow, and potential tools. We discuss the results for each group separately within each section.

3.2.1 Results: About the Participants

The first section of the survey established characteristics of the developers. We wanted to know about their current and past experience with assembly as well as other programming languages. We also wanted to establish their use of tool support in general.

Mainframe Context

Table 3.2 summarizes the profiles of our mainframe respondents. It is important to note that for some of these questions, respondents could provide more than one answer (i.e. favorite tools). We found that the majority of people had over 10 years of experience and were most familiar with assembly language. They also preferred assembly language over others in many cases. Their favorite software tools fell under two categories: debuggers and text editors. There were nine who responded with some type of text editor, which included Emacs, VI, KEdit, XEdit, UltraEdit and more. Of those who mentioned a debugger, XDC was mentioned most often.

In addition to these results, we also asked why these programming languages and tools were their favorite. With regard to languages, assembly was preferred because it is *easy, elegant, maps to hardware, powerful, efficient, simple/intuitive, versatile,* and *fun*. REXX was also listed as *fun, easy* and *had string handling*. C/C++ has *better abstractions, easier syntax, portable* and *simple*. Python was mentioned as *having a good API* and *fits all paradigms*. Though text editors were mentioned more often, no particular reasons were given other than the text editor they chose was configurable and customizable. Debuggers were mentioned for *tracing instruction flows*, and *examining/manipulating the program environment (e.g. data/registers)*. Visual Studio was mentioned because it *has a debugger and has syntax checking*. Eclipse also was mentioned for *being so well integrated with the Java language*.

Malware Context

In the malware context, Table 3.3 outlines the profiles of our respondents. Again, most of the respondents had over 10 years of experience, and while they may work often with assembly, they are more familiar with C/C++ and Java, and also only prefer C/C++, Java and Python. In fact, these were the only provided favorite programming languages, so no respondent actually preferred assembly. Their favorite

Asked	Reported
Experience	88% (22/25) 10+ Years
Most Familiar Programming Language	100% (25/25) Assembly 48% (12/25) REXX 48% (12/25) C/C++ 16% (4/25) COBOL
Favorite Programming Language	56% (14/25) Assembly 16% (4/25) REXX 16% (4/25) C/C++
Favorite Tools	36% (9/25) Text Editor 32% (8/25) Debugger 16% (4/25) ISPF

Table 3.2: About the Mainframe Respondents.

tools included IDEs such as IDA Pro, Eclipse and Visual Studio. In contrast to above, only a few listed a text editor and/or debugger.

Their reasons for preferring these languages varied. For Java, it *avoids memory problems, has an API, object-oriented* and *has easy network communication.* For C/C++ they said it *had the most control, was more flexible, fast execution device drivers, could be embedded on multiple platforms* and *had memory management.* Python was mentioned as being *fast, good for prototyping* and *simple.*

In regard to tools, IDA Pro was stated by multiple respondents to be the *best disassembler.* It was also stated that it is a *wonderful static debugging tool*, and is *very scriptable and extendable.* Though it was also mentioned that the *extensibility is poorly documented.* One respondent also said that it has *features not found in any other tool.* Eclipse was preferred due to the *abundance of plugins, cross-platform, code completion and refactoring, debuggers, that it has a community*, and *supported large software projects.* Visual Studio was mentioned under the same umbrella as Eclipse with *code completion, refactoring*, and *good debuggers.* NetBeans was also mentioned because it is *easier to setup than Eclipse.*

3.2.2 Results: Assembly Experience

The second section of the survey established how comfortable our respondents were with assembly language itself. This was important in knowing whether the tool support we build should be geared towards novices or experts. We also wanted to know in which context it was used, and what, if anything, is difficult or time consuming about working with assembly.

Asked	Reported
Experience	79% (11/15) 10+ Years
Most Familiar Programming Language	93% (14/15) C/C++ 67% (10/15) Java 47% (7/15) Assembly 27% (4/15) Python
Favorite Programming Language	47% (7/15) C/C++ 40% (6/15) Java 20% (3/15) Python
Favorite Tools	47% (7/15) IDA Pro 40% (6/15) Eclipse 33% (5/15) Visual Studio 20% (3/15) Text Editor

Table 3.3: About the Malware Respondents.

Mainframe Context

Table 3.4 summarizes the results for respondents' experience. With regard to experience with assembly itself, developers reported being slightly less adept at writing assembly than they were at understanding it. They were, however, very sure of themselves on both, averaging at 4.42 and 4.46 respectively on a 1 - 5 Likert scale [80]. As for whether assembly was more difficult to understand than other languages, 16 respondents thought that this was true. Of those that said it was more difficult, six said it was because *you had to really understand the instruction set for which there were many operations and rules*; five said it was because it was *hard to see a high-level picture since there were so many lines for a simple task, and that also meant the developer had a lot to remember at once.* Finally, four said it was because you had to have a *thorough knowledge of the underlying hardware or operating system.* Of those that said it was not more difficult, there was no convergence on why, although some reasons were that it *depended on the quality of the code written, that each instruction was overall simpler (although more tedious)*, and it was *easier to find bugs or performance problems.* Of the 18 respondents who thought there were more difficult languages, the majority (7 respondents) thought that C/C++ was the most difficult because of the *more difficult syntax, method overloading and multiple inheritance, the preprocessor, and memory issues such as leakage*, and *controlling pointers.* Other languages worth mentioning are COBOL and LISP.

The most familiar assembly language mentioned was HLASM by 20 respondents. In terms of what assembly was used for, 18 used it for development and 7 mentioned

maintenance. This development included operating system changes, databases, monitoring and language support. Other types of development included times when high performance was critical (financial uses) and for security and encryption. Finally, three respondents used it while debugging and only one used it for reverse engineering.

They were also asked what the most difficult task they had to perform was as well as which took the most amount of time. The top reported most difficult tasks were testing, debugging, understanding code written by others, and understanding new systems. Multi-threaded applications were mentioned, as well as performance analysis, application structure and finding out which path through the code had been executed. Equally reported were the tasks of figuring out specifications and documenting results. These were also the highest-rated, most time-consuming tasks.

Asked	**Reported**
Experience writing	4.42 (out of 5) / 0.97 Standard Deviation
Experience understanding	4.46 (out of 5) / 1.06 Standard Deviation
Is assembly more difficult?	64% (16/25) Yes
If so, why?	38% (6/16) Many low-level operations 31% (5/16) Big picture obscured 25% (4/16) Knowledge of underlying hardware/OS
Are any more difficult?	25% (6/24) No 29% (7/24) C/C++ 12% (3/24) COBOL 8% (2/24) LISP
Most familiar dialect	80% (20/24) HLASM
How assembly is mostly used	78% (18/23) Development 30% (7/23) Maintenance 13% (3/23) Debugging
Most difficult task	19% (4/21) Testing 19% (4/21) Debugging 19% (4/21) Documentation 14% (3/21) Understanding others' code 10% (2/21) Understanding new systems
Most time-consuming task	25% (5/20) Testing 20% (4/20) Debugging 20% (4/20) Understanding others' code 15% (3/20) Documentation 10% (2/20) Understanding new systems

Table 3.4: Assembly Experience of Mainframe Respondents.

Malware Context

Table 3.5 summarizes the results for assembly experience. Respondents in this context reported being slightly less adept at writing assembly than they were at understanding it, averaging around 2.9 and 3.5 respectively on a 1 - 5 Likert scale. There were 12 respondents who thought assembly was more difficult to understand than other languages. The reasons for this included that there were *so many low-level operations that each do not do much, hard to see the big picture, not high-level so hard to translate* and *having too many coding conventions*. Other reasons included the *limited number of registers in x86 which caused many copy/reuse instructions so it was hard to see real variables, logic conditions less apparent, following control flow of instructions, data flow (may effect other registers and side effects of instructions), having to keep the stack in your head*, and that *optimizations change the code*. We also asked if there were more difficult languages than assembly. Functional programming languages as well as LISP and Prolog were identified.

Every respondent mentioned x86 as being one of their most familiar assembly dialects, though seven respondents also mentioned another, including ARM, and Power PC. They used assembly mostly for malware understanding and second for program understanding, which was defined as *including embedded systems*, as well as *looking for security holes*. In this manner, these two may fall under the same category. Reverse engineering was also mentioned both for software as well as for devices. Other answers included *comparison of malware families, core dump understanding, performance bottlenecks, hand optimization* and *debugging in "release only"*.

They were also asked what the most difficult task they had to perform was, as well as which took the longest. The top reported most difficult task was *control flow*, followed by *data flow, deobfuscation* and *decryption*. Although many other tasks were mentioned, including *documentation, getting the high-level picture, debugging* and *process communication*, the most time-consuming tasks were *locating certain behaviour within the code, control flow analysis, data flow analysis, deobfuscation* and *decryption*. Again, the same tasks were noted in the most difficult task section.

3.2.3 Results: Current Tools

This third section of the survey aimed to find out which tools are currently being used by each group. We wanted to know those they used to work with assembly, and what their strengths and weaknesses were.

Asked	Reported
Experience writing	2.9 (out of 5) / 0.92 SD
Experience understanding	3.5 (out of 5) / 0.74 SD
Is assembly more difficult?	80% (12/15) Yes
If so, why?	33% (5/15) Many low-level operations 20% (3/15) Big picture obscured 13% (2/15) Translation to high-level 13% (2/15) Reliance on conventions
Are any more difficult?	47% (7/15) No 33% (5/15) Functional PLs 7% (1/15) Prolog
Most familiar dialect	100% (15/15) x86
How assembly is mostly used	47% (7/15) Malware understanding 33% (5/15) Program understanding 20% (3/15) Reverse engineering
Most difficult task	27% (4/15) Control flow 20% (3/15) Data flow 13% (2/15) Deobfuscation 13% (2/15) Decryption
Most time-consuming task	20% (3/15) Locate behaviour 13% (2/15) Control flow 13% (2/15) Data flow 13% (2/15) Deobfuscation 13% (2/15) Decryption

Table 3.5: Assembly Experience of Malware Respondents.

Mainframe Context

Table 3.6 summarizes the results for respondents' current tool use. The tool primarily used by the majority of developers (17) for working with assembly language was some kind of text editor. We then asked what other tools they used besides the primary one. In this case, 15 respondents mentioned a particular debugger, XDC being the one mentioned most often. Other tools mentioned were being able to do *diffs, greps* and *view traces.*

As for deficiencies with these tools, eight respondents said that there were none, but five wanted extra features in their text editor such as syntax checking and highlighting. Three respondents wanted to be able to *navigate more easily by following cross references.* Two respondents mentioned some sort of diagram support including *trace/flow or code diagramming tool*, and one mentioned being able to *integrate with a modern GUI with tools allowing querying about the code.*

When asked what the best features of these tools were, five people responded that they were interested in data, for example, *what contents are in a variable or register*

Asked	Reported
Primary tool	68% (17/25) Text editor 12% (3/25) HLASM Assembler
Secondary tools	60% (15/25) Debugger
Deficiencies	33% (8/24) None 21% (5/24) Text editing (syntax highlighting, checking) 13% (3/24) Navigation within code
Best features	31% (5/16) Data (register/var contents, memory/data flow) 19% (3/16) Single step execution 13% (2/16) Syntax highlighting 13% (2/16) Trace or Dump output

Table 3.6: Current Tool Use of Mainframe Respondents.

and how that data is used throughout the system (data flow). Single step execution was most important for three respondents, and two found syntax highlighting the most beneficial and two mentioned being able to *see a trace or dump file.*

The final question in this section was in regard to IDA Pro. This question was included for the malware context. None of the respondents in this domain used it, and one person *may have heard of it although understood it did not work with mainframe assembler.*

Malware Context

Table 3.7 summarizes malware tool use results. The tool primarily used by 13 malware analysts was IDA Pro. Secondary tools included different hex editors, WinDbg, various IDA Pro plugins, as well as other various debugger, memory analysis and simulation tools. Table 3.7 outlines the tools used and their strengths and weaknesses.

The most reported deficiency with current tools was that there is *no integration between them, no way to link their best features.* Other reasons included *providing definitions of instructions, documentation (for example to be able to add notes)*, and *converting to C.* Other requests included *having assembly debugging in Eclipse, to be able to take a snapshot of a debugging session and restart execution to a particular location, customize IDA Pro's graph view*, as well as *IDA creating better function names based on hierarchy.*

They were also asked what the best features of the tools they used were. The graph view in IDA was mentioned first, followed by the extensibility of IDA Pro through plugins, search patterns in IDA Pro, and being able to *inspect and modify registers, stack and heap space.* PaiMei was mentioned due to being able to *quickly*

Asked	Reported
Primary tool	87% (13/15) IDA Pro
	7% (1/15) PVDasm
	7% (1/15) NASM
Secondary tools	33% (5/15) Hex editors (e.g. 010)
	27% (4/15) WinDbg
	20% (3/15) IDA Pro plugins
Deficiencies	20% (3/15) Lack of integration
	13% (2/15) Instruction assistance
	13% (2/15) Documentation
	13% (2/15) Convert to higher-level
Best features	20% (3/15) IDA Pro Graph View
	13% (2/15) IDA Pro extensibility
	13% (2/15) IDA Pro search patterns
	13% (2/15) Inspect and modify heap/registers/stack

Table 3.7: Current Tool Use of Malware Respondents.

and dynamically pinpoint functions.

We further asked which IDA views they used most frequently, already knowing that IDA Pro would be the target platform for this group. By far, 10 respondents mentioned the graph view, 9 mentioned the text view, 5 mentioned the hex view and the function view. After that the most popular in decreasing order were registers, names, strings, and imports.

It is important to note that the industry proposes other good tools for malware analysis such as Norman's Sandbox Analyzer Pro [81], Sunbelt's CWSandbox [82] and HBGary's Responder [22]. These tools, however, are usually extremely expensive (in the tens of thousands of dollars (USD)) and were out of reach for our respondents.

3.2.4 Results: Browsing and Navigation

This was the fourth section of the survey. When trying to form a bird's eye view of the system, it is important to provide varying degrees of granularity. For this reason, we targeted questions as to what beacons exist in assembly. A *beacon* is a recognizable, familiar feature in the code that acts as a cue to the presence of certain structures. Beacons are used to move from high-level abstractions or concepts to lower-level details [83].

Asked	Reported
Beacons	76% (19/25) Specific instruction 16% (4/25) Comments 16% (4/25) Macros 16% (4/25) Loops 16% (4/25) None
Task-focused interface	36% (8/22) No 32% (7/22) Yes 32% (7/22) Unsure
Zoom	15% (4/25) Do not have long modules 29% (5/17) Subroutines 12% (2/17) Macros 12% (2/17) CSECTs
Additional features	33% (5/15) Follow links (branches, cross references, declarations)

Table 3.8: Browsing and Navigation for Mainframe.

Mainframe Context

Table 3.8 summarizes the following results. The first section dealt with browsing and navigation, and to see which cues we can use for navigation. The majority, 19 respondents, mentioned a specific instruction, BALR being the most common, which is used to make subroutine calls. Other important responses were comments, macros and loops, mentioned by four respondents each. Finally, four respondents thought that there were no beacons at all.

The next question asked whether they could benefit from a task-focused interface. The task-focused interface is a type of user interface which extends the desktop metaphor of the graphical user interface to make tasks, not files and folders, the primary unit of interaction. A well-known example of this would be Mylyn [84]. The results were split almost equally; eight thought no, seven said yes, and seven were unsure. Judging by the responses, it was not clear what the idea of a task-oriented interface actually meant.

We then asked if there were artifacts in the code that would be useful to zoom into a higher level of detail. There were eight who did not respond, or said it was not applicable, with four respondents mentioning that it was policy not to write long modules, so zooming was not necessary within a module. Of the 17 that did respond, the majority mentioned subroutines, including separating OS functions and external subsystems from application code. Macros, CSECTs, DSECTs and dynamic storage areas were mentioned but only by a minority.

They were lastly asked for any general comments on this topic, the majority of those that answered wanted to be able to follow "links" in the source. Such links would include branches and cross references, but also being able to jump to declarations and return. One of the answers mentioned that something like CTAGS would be useful, which provides different linking depending on the language, but with a good idea of what items to link.

Malware Context

Table 3.9 summarizes the results for beacons and other aspects that could be used for navigation. Possible beacons given by this group included control flow features such as function calls, and branching. Ways in which data is used were also mentioned, tied with code conventions such as load/stores to indicate the start of functions.

As for whether a task-focused interface would be useful. Of the six who responded, all said that yes, it would be useful. We propose that those who did not respond were unsure and did not understand the concept.

We also wanted to know which artifacts existed in the code that they thought could be used as means of navigation by zooming. Functions were mentioned foremost, followed by modules. Also mentioned were basic blocks, instructions, and blocks of memory locations.

As for comments on other features they would like for browsing/navigation, a history view that could show all paths taken, color-coded by the number of times, was mentioned twice. The following were also mentioned: *having a two column viewing showing callers and callees, navigation between high-level and low-level details, customizable interfaces, sequence charts*, and *data flow*. These were only mentioned by one individual however, so we do not show them as an aggregate in the table.

Asked	**Reported**
Beacons	27% (4/15) Function Calls (Control Flow) 27% (4/15) Data Usage 27% (4/15) Coding Conventions 20% (3/15) Function Definitions
Task-focused interface	100% (6/6) Yes
Zoom	40% (6/15) Functions 20% (3/15) Modules

Table 3.9: Browsing and Navigation for Malware.

Feature	$\overline{\chi}$	SD
Where is a particular subroutine/procedure invoked?	4.44	0.92
What are the arguments and results of a function?	4.76	0.66
How does control flow reach a particular location?	4.68	0.85
Where is a particular variable set, used or queried?	4.6	0.76
Where is a particular variable declared?	3.76	1.16
Where is a particular data object accessed?	4.28	0.98
What are the inputs and outputs of a module?	4.44	0.92

Table 3.10: Importance of Debugging Features for Mainframe.

3.2.5 Results: Debugging

In this fifth section of the survey, we wanted to see how important debugging was. In order to do so, we asked about existing theories of comprehension about debugging [85] and their importance with respect to assembly. Each aspect could be given a score on a Likert scale of 1 - 5.

Mainframe Context

We show the mean response as well as standard deviation in the order presented in [85] in Table 3.10. Many of the respondents simply answered five for all, which resulted in a similarly high rank for many items, while only one item scored below four.

We then asked if there were any other features of debugging that were important with regard to assembly. Eight respondents mentioned something to do with data flow. This included *where a data object or storage was written*, and *how registers pertained to storage*. Another somewhat common answer was to do with variables, where they were used, how they were declared and if they were ever used in a way they should not be.

We then showed the mockup of a potential debugging tool that resulted from interviews with mainframe developers. This tool is shown in Figure 3.1. The general reaction was very positive. There were a few suggestions made that will definitely need to be taken into account. That is the addition of extra data information. Currently we only show the values in the registers, however showing the current module's storage, including variables and data structures, was mentioned by six of the respondents. Showing a distinction between the module and CSECT, and to show DSECTs as well, was also mentioned by two respondents.

Feature	$\overline{\chi}$	SD
Where is a particular subroutine/procedure invoked?	4.80	0.41
What are the arguments and results of a function?	4.53	0.83
How does control flow reach a particular location?	4.60	0.51
Where is a particular variable set, used or queried?	4.60	0.63
Where is a particular variable declared?	3.53	1.51
Where is a particular data object accessed?	4.33	0.82
What are the inputs and outputs of a module?	4.13	0.92

Table 3.11: Importance of Debugging Features for Malware.

Malware Context

Table 3.11 shows the importance of the same debugging features, on a Likert scale of 1 - 5. These results show that the top two most important features pertain to control flow, further highlighting its importance in this domain. Again, the same item fell below the score of four.

The responses for any missing features of debugging in assembly were extremely varied but included *data flow, doing trace "diffs", memory view, stepping back in debugging to see how a var value was assigned, setting register/variable values and re-running, access list to specific memory address, simulate execution statically, multi-application debugging (malware and the infected file)*, and finally *using a standard so all tools can communicate.* One area of concern we see in a few sections of the survey is the ability to change input values and run the system again.

Since we distributed the same version of the survey to both the mainframe and malware groups, the mockup debugging tool for mainframe assembly was also shown to this group. None of the responses were positive, with most respondents not understanding the diagram, or not believing it would be useful. We therefore conclude that this particular form of a debugging tool is only useful within the mainframe context.

3.2.6 Results: Control Flow

The sixth section of the survey was dedicated to control flow, which we knew from previous conversations with stakeholders was a huge issue for those working with assembly.

Mainframe Context

Table 3.12 summarizes the results for control flow. The first question asked about static control flow scenarios, of which our Tracks tool (Section 6.3) has two. The first is static control flow that shows all of the functions that could possibly be called from a specific function. It also provides navigation support to the function in IDA Pro through double-click. The second is a history view which diagrams the functions the user has navigated to within IDA Pro. We asked if the developers could see any other static scenarios. The only comments here pertained to dynamic control flow, suggesting that static control flow may not be of interest.

We were also curious how useful a reversed control flow would be. Reversed control flow means given a function, you can step backwards to see what paths led there. We asked how useful each was on a Likert scale of 1 - 7. We used a 7-point scale for greater accuracy on this question because it is not asked in other ways elsewhere. Forward control flow was rated at 5 and reversed at 5.08. We also asked if it was useful as a yes or no response, and 19 respondents said that it was useful. We then asked when people would use this reversed flow, most answered as we expected, which is, once a point of interest is identified, see what paths led there. Others included *looking at a dump file during post-mortem analysis* and *assessing impact of a change.*

Next we looked at dynamic control flow, of which Tracks has one scenario. We can open a dynamic control flow diagram on a function and when it is opened, a breakpoint is set in IDA Pro at that function. When a breakpoint is reached, users can step through the code and each new function reached is added to the diagram. There is also the option to diagram all calls even if they were not stepped into. This information can then be saved to a trace file. We asked if the developers could see any other dynamic scenarios. Two respondents mentioned being able to *see paths of execution taken most often to identify hotspots in the program*, and also *see common loops and flow.*

Finally, we asked what information developers might want to mine from control flow. There were three areas that stood out. The first was with concern to data flow, *what values were in the registers when a subroutine was called and when we returned (parameter values), as well as register values at interrupts.* Additionally, *how base registers are mapped and who modified what memory and when.* The second area of interest had to do with analyzing performance by using timing information with the dynamic trace. Finally, being able to mine statistics about the subroutine and system

Asked	Reported
Static concerns	None
Dynamic concerns	8% (2/25) Most executed paths
Forward control flow	5.0 (out of 7) 1.66 SD
Reversed control flow	5.08 (out of 7) 1.73 SD
Reversed flow useful	90% (19/21) Yes
Information to mine	24% (6/25) Register values/mapping, memory usage 12% (3/25) Performance 12% (3/25) System and subroutine call statistics

Table 3.12: Control Flow Requirements for Mainframe.

calls. These statistics might be used to identify code that could be refactored into a subroutine or macro.

Malware Context

Table 3.13 summarizes the following results. The first control flow question was again about static control flow. We asked if they could see any other static scenarios. They did not list any but did show concern for how loops, recursion and random locations in the function table would be handled.

We also wanted to know how useful reversed control flow would be. We asked how useful each was on a Likert scale of 1 - 7. Forward control flow was rated at 6.4 and reversed at 6.15. We also asked "yes or no, is reversed flow useful?", and 13 responded said that it was.

Next we looked at dynamic control flow. We asked if the developers could see any other dynamic scenarios other than an execution trace. They did not, but expressed that these dynamic diagrams should be able to handle multi-threaded traces and show when execution switched between IDA Pro instances. They also wanted to be able to compare two traces, and see how often a branch was taken to see which branches are critical.

Finally we asked what information developers might want to mine from the control flow data. Mentioned most was call patterns, as well as API call patterns, and network/communication patterns. Again, being able to do a diff/compare of traces, and looking at what data is used to determine branching was mentioned.

Asked	Reported
Static concerns	20% (3/15) Loops and recursion
Dynamic concerns	Multi-threaded traces Compare traces Branch frequency
Forward control flow	6.38 (out of 7) 0.87 SD
Reversed control flow	6.15 (out of 7) 1.07 SD
Reversed flow useful	87% (13/15) Yes
Information to mine	47% (7/15) Call patterns 13% (2/15) Compare traces 12% (2/15) Reach execution points (jump conditions)

Table 3.13: Control Flow Requirements for Malware.

3.2.7 Results: Potential Tools and Wish List

The final section of the survey asked the respondents specific questions about existing IDE features, as well as our proof of concept and other potential tools.

Mainframe Context

First we asked how useful developers would find the standard IDE features in regard to assembly on a scale of 1 to 5. The results are listed in Table 3.14. We can tell from these results that most people in this domain are actually writing assembly code, as well as trying to understand it.

Asked	Reported
Syntax Highlighting	3.52 (1.42 SD)
Syntax Checking	3.48 (1.36 SD)
Code Completion	2.56 (1.29 SD)
Search for References	3.96 (1.31 SD)
Go to Declaration	3.88 (1.20 SD)

Table 3.14: Importance of IDE Features for Mainframe.

We then asked about a tool prototype called LegaSee (based on the AJDT Visualiser) that we had developed after discussing issues with mainframe developers. This tool is further discussed in Section 6.4. Developers were asked to comment on its usefulness, and 10 thought that it would be useful (some even commented very useful), 8 had no comment, 4 were unsure and only 3 said no. We also asked for any other comments and the one thing that stood out is that developers wanted to be able to see from which modules or CSECTs, that DSECTs were actually referenced from,

and also if they were referenced or updated. We also asked what parts of the system would be useful if we wanted to be able to zoom into the system to see more detail. Five of the respondents replied subroutines, which included differentiating between local calls, OS calls and external subsystem calls, while four said they never have to work with long modules, and one mentioned using macros, which included site-specific macros. This means that the type of macros used would need to be customizable in the tool. One example included macros at the start and end of functions which had provided issues for other similar tools. Finally, one respondent mentioned showing the *DSECTs and storage used being linked to where they are used in this diagram.*

The next question asked users about a tool called MapUI that is discussed further in Section 2.2.4. Only five respondents thought there may be a use for this tool, and seven were unsure. It was mentioned that the size of the countries should be configurable, for example, their size in lines of source code might not be important, but their size in number of other subroutines that call it might be very useful.

The next question asked how users felt about a split view with assembly on one side and a high-level language on the other, such as C. This would give a beginning assembly programmer a better idea of what was going on. Of course in this domain, the developers have a thorough understanding of the assembly. Only five respondents said it would not be useful, but the others had taken it to mean that the assembly view would only be to aid with the development of the higher-level language, in almost all cases, the language mentioned was COBOL.

An interesting question that we asked last is since sequence diagrams have shown promise, are there other UML diagrams that might also be useful? It seems here that most people were not sure what UML diagrams were since 11 did not answer, and 5 said they were unsure. Only one respondent mentioned state diagrams and one class diagrams.

The very last section asked if there were any other items on their wish list for assembly. Four respondents wanted something for pattern recognition. The reasons varied for this, but included to *refactor redundant code, see the relationship between naming conventions in labels*, and *see which code accessed the same variables (or data).* Here we also note three other areas that were seen frequently in responses. The first is *being able to have a better macro processor and language as well as visualization of macros.* Second is to *have better debugging including better breakpoints.* Finally having a *better profiler to improve performance.*

Malware Context

As far as rating what most high-level IDE tools provide, the most important feature was finding references, followed by finding the declaration and syntax highlighting. Syntax checking and code completion were noticeably less important, likely due to the fact that this group focuses on understanding existing code, rather than creating new code. The results are shown in Table 3.15.

Asked	**Reported**
Syntax Highlighting	4.40 (0.74 SD)
Syntax Checking	3.33 (1.29 SD)
Code Completion	2.93 (1.10 SD)
Search for References	4.73 (0.46 SD)
Go to Declaration	4.60 (0.63 SD)

Table 3.15: Importance of IDE Features for Malware.

We then asked about the LegaSee prototype that had been developed in response to an observed issue with the mainframe group. We wanted to see if there was a perceived potential from this group as well. We asked if they thought this type of tool could be useful. Only one of the respondents could see a use, with the majority not understanding the concept behind it. We conclude that within this particular domain, LegaSee may have no use.

The tool we asked about next was the MapUI software terrain map tool. This tool was initially developed with feedback from members within this survey group. When asked if it was useful, five thought that it was useful, while three were unsure. Of those that thought it had a use, function interaction was mentioned, as well as size and relationships of modules. The ability to combine this view with tasks was mentioned as well.

As for the usefulness of a split view with assembly and another higher-level language, 11 thought it was useful. One comment made was that it would be good for beginners. Two respondents mentioned that they already used this feature within the Visual Studio debugger, and the Hex-Rays decompiler plugin for IDA Pro.

In regard to further UML concepts, five of the developers felt that state diagrams would be useful, and two thought that activity diagrams would be useful. A lone response suggested package diagrams with packages representing modules.

The final question was open ended and asked about any tools on their wish list. Only one item was listed by two respondents which was *better integration with other*

tools, one person mentioning *using meta-assembly to push back and forth with other tools*. The rest were only mentioned once and included: *data flow, sequence viewer, pattern recognition, documentation, creating C from ASM to guess what a function is doing*, and *omniscient debugging*.

3.2.8 Survey Summary

This exploratory survey has looked at seven factors: respondent experience and preferences with programming in general, familiarity with assembly language, their current toolset for assembly, browsing and navigation concerns, features of debugging, control flow, and a look thus far at some potential tools. The following two subsections summarize the key points for each group with respect to these factors. We further compare the results of these two groups in Chapter 5. In an effort to condense the information shown in previous sections, we organize the most salient results and display them for both groups in Appendix A.

3.3 Collaboration and Documentation for Malware

We invited malware analysts by email, to take a small survey to find out how documentation and collaboration are currently used and whether or not their current tools are sufficient. The emails were sent to people we had contact with, who were able to freely forward it to others they thought would be interested. We had six respondents, all reverse engineers in industry or academia. The survey was issued via Google Forms, contained 11 questions in total, and was expected to take 10 minutes to complete. The results are discussed below for each topic, and a summary and analysis are provided at the end of this section.

3.3.1 Results: Collaboration

This part of the survey contained six questions that first asked whether or not there was a need to collaborate, how the collaboration occurred and what tools were currently used for that collaboration. Finally, we asked whether or not these tools were sufficient and what features they would like to see. The results for each question are discussed in this section.

When you're working at the assembly level, do you have a need to collaborate with others? This question was given as a simple yes or no answer, with four respondents answering yes, they did collaborate, while two did not.

If yes, how do you collaborate? Of the four respondents who did collaborate, they did so by talking to each other (including giving each other crash courses on what they had done) and sharing information. In one case, this was achieved by working on the same computer.

For document sharing, three mentioned that they shared IDA Pro [5] database files (IDB files) which included documented pieces of code, whereas two mentioned sharing information from other sources as well, such as documents, internet links, and reports.

If yes, who do you collaborate with? Do you collaborate with them all at once? Of the four out of six respondents who collaborated, all of them did so with co-workers (members of the same team) or co-researchers. All of them collaborated in small groups, with one or two colleagues at a time, and three respondents also collaborated all at once in a large group. One respondent mentioned that the large group would be to share end results, so it may be that the group size is dependent on the collaboration intent.

If yes, what tools do you currently use to support your collaboration? Of the four respondents, three did not mention any software tools, though all respondents did use devices such as phones, presentations on projectors, email for sharing documents as well as memory sticks and shared folders. There was one respondent who had tried CollabREate, but the tool was not usable in practice since their malware analysis required them to be on isolated computers not connected to the internet for security purposes.

Are these tools sufficient? Again, this was a yes or no answer, and five respondents answered this particular question. Four said that no, the tools are not sufficient while one said that they were.

If not, what features do you wish you had for collaboration in your tool environment(s)? Or if you don't use any tools, what features do you wish you had? Answers to this question varied but answers included:

- synchronize documentation efforts by importing pieces that were already documented
- share comments on specific parts of code
- share renamed variables and functions
- retrace steps that an analyst did that were useful
- status updates on what an analyst just did or found, such as "X just renamed sub_0000ABCD to XYZ"

3.3.2 Results: Documentation

This part of the survey contained five questions, asking whether or not there was a need to create documentation and if so, what kind of documentation and what tools were used to create it. We then asked whether or not these tools were sufficient and what features they would like to see. The results for each question are discussed below.

When you are working at the assembly level, do you have a need to create documentation? This was a simple yes or no answer, and all six respondents said that yes, they do create documentation.

If yes, what form does the documentation take? If you can, please post a sample in the box below. Otherwise please comment on what a sample of documentation would look like. The most prevalent answer to this question was comments within IDA Pro which four respondents mentioned. There was one respondent who mentioned that they rely on making the comments appear semi-Javadoc so that they are easier to parse, and one respondent who also mentioned that comments were made on paper.

There were four respondents who mentioned analysis summaries within reports while two respondents mentioned that they created these reports with code snippets of the disassembly. One respondent also added that a sequence diagram might be included in the report to show the call sequence leading to the problem.

If yes, what tools do you currently use to support your documentation? While two respondents reported using no tools at all, an equal number used Microsoft Word, Excel and OneNote. One use of Excel was to create control flow graph representations using hyperlinks. Tools mentioned by one respondent each included LaTeX, Visio, OneNote, IDA Pro, and a WIKI.

Are these tools sufficient? Given as a yes or no option, five respondents said that no, the tools were not sufficient while one said that they were.

If not, what features do you wish you had for documentation in your tool environment(s)? Or if you do not use any tools, what features do you wish you had? There were four responses to this question, with two respondents mentioning being able to integrate information from various sources, including from reverse engineering tools. This would include pulling information from different sources (i.e. IDB files, API documents, diagramming tools) to consolidate into one report, and one respondent mentioned that this may be possible through tagging.

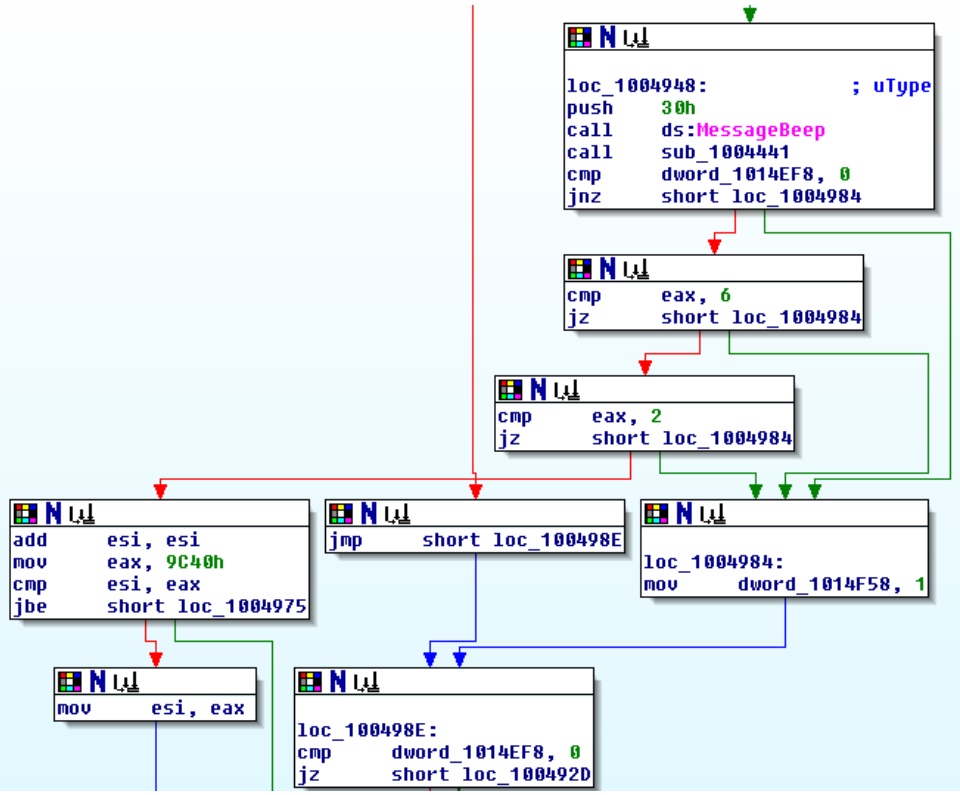

Figure 3.2: The Graph View in IDA Pro.

The need for comment support was discussed by two respondents. One discussed tagging and grouping of comments as well as searching for them. The tagging would allow comments on specific code, basic blocks and variables. The other respondent mentioned having to annotate a snapshot of the graph view in IDA Pro. This graph view is shown in Figure 3.2. This respondent also mentioned the need to make

Collaboration
67% (4 out of 6) is needed 80% (4 out of 5) current support is insufficient Features requested: Share existing documentation Retrace others' steps Status updates on analysts' actions
Documentation
100% (6 out of 6) is needed 83% (5 out of 6) current support is insufficient Features requested: Better comment support Better documentation integration Document execution paths

Table 3.16: Survey of Malware Collaboration and Documentation.

integration automated by being able to export comments to the documentation system so that it remains up-to-date when changes are made in IDA Pro.

There was one respondent who mentioned being better able to document execution paths, as well as creating small videos to explain concepts that will need to be repeated, such as how we reached a location and the path that led there and why a function is being renamed.

3.3.3 Results: Summary

Table 3.16 shows a summary of the results from this user survey. Collaboration was reported as necessary by four of the respondents, however four also reported that their tools were not sufficient. The features requested included being able to share existing analysis documentation, including comments and renamed variables and functions, retrace steps that another analyst took as well as follow status updates on analysts' work.

As for documentation, all six respondents needed to create documentation while five reported that the tools were insufficient. Respondents wanted better support for comments within the reverse engineering tools, better integration between the different documentation tools, and the ability to document execution paths.

3.4 Chapter Summary

This chapter provided the results of interviews and surveys with both mainframe developers and malware analysts. Within the mainframe group, interviews took place with a select few individuals and were presented by each interview. The large exploratory survey that was issued to each group was identical, and the results for each group are discussed and presented by each section of the survey. We summarize all results in Appendix A for each group, respectively. Finally, we presented the results of a survey conducted with a small group of malware analysts on the specific topics of collaboration and documentation.

The following chapter provides the results of the requirements elicitation performed with each group. Chapter 5 then further evaluates these results, by comparing and contrasting the groups' profiles and issues.

Chapter 4

Requirements Elicitation

The previous chapter, in addition to related work [86, 87], has shown that current tool support for the systems we are investigating is inadequate. While our exploratory survey aided us in forming a preliminary view of the problem space, we still need to elicit requirements to drive the future development of assembly language comprehension.

This chapter reports the results of a requirements elicitation study that took place at two industrial software teams [88]. The first of these teams falls under the mainframe development category. The second team represents the malware analysis perspective. We will discuss the method we used for the study, as well as requirements elicited from each group.

4.1 Elicitation Method

This section introduces the two groups that we study, as well as each step of the elicitation process. The background and motivation behind the method used is provided in Chapter 2.1.

The requirements elicitation process we followed can be split into the following steps. Each of these steps is described in order within this section. Steps 2 and 5 required us to meet face-to-face with participants, and were therefore not possible with the malware group.

1. Identify the profiles of participants through surveys.
2. Activity-based protocol elicitation (mainframe group only).
3. Normative manipulation techniques to increase information sharing.
4. Nominal group technique to elicit requirements.

5. Exit interviews (mainframe group only).
6. Exit survey to capture final opinions.

4.1.1 Elicitation Setting

The mainframe group consists of seven co-located team members of one company working in the area of assembly language programming on the company's mainframe. The team belongs to a large multinational company which employs more than 13,000 staff. Every member of the group is a long term employee in the area of relevance.

The malware group is in a national military research department. The team consists of eight members that investigate how to create more secure software and better secure existing software. They use reverse engineering tools and techniques to provide hindsight on malware and security flaws. Additionally, in contrast to the mainframe group, the malware group has been a stakeholder in our Assembly Visualization and Analysis (AVA) project from the beginning. We do not see this having affected our results, but it is a factor to consider for those looking at performing this elicitation with other unique teams.

The subjects in each group were not made aware that normative manipulation was carried out for the duration of the study. For each group, an initial survey was distributed to the team in advance to perform user profiling. This was then followed by normative manipulation and requirements elicitation, and finally an exit survey.

Ethics approval for this study is shown in Appendix F.

Mainframe Group Process

Previous to arriving, we distributed the user profile survey by email weeks in advance. We then spent three mornings on site. On the first morning, we observed two employees and distributed the individual priming and requirements elicitation exercise to each team member. On the following morning, we led the nominal group session based on this exercise. On the final morning we conducted exit interviews with two team members to discuss the meaning of requirements produced by the group session. Finally, we issued an online exit survey by email to capture thoughts about the process one week after our visit.

Malware Group Process

It was not possible to be on-site with the malware group due to travel constraints. Furthermore, observational studies would not have been allowed due to the malware group's nature as a security facility. Therefore, we performed all of the same elicitation steps, except the activity-based elicitation and exit interviews. The user profile survey, and the individual priming and requirements elicitation exercise, were completed over email. The nominal group session was conducted over video conference, with the malware group located together face-to-face. Finally, the exit survey was sent by email invitation within a week of the group session.

4.1.2 User Profiles

The user profile survey was administered online as double-blind with the LimeSurvey platform [89]. This initial survey had three goals: to determine how the group interacts and how knowledge is distributed, to obtain an NFC measurement for each individual, and finally to determine their INDCOL score. After the profiles of the participant groups were determined, the normative manipulation appropriate to the group type could be applied.

NFC and INDCOL Measures

The original NFC scale developed by Webster and Kruglanski [90] contained 42 items and included five facets. Due to constraints on participants' time, the full NFC scale was not used. Instead, we used a revised version, developed by Roets and van Hiel, that contained 15 items [91]. The revised version does not allow for the assessment of each facet of the scale, but provides an overall measure of individual differences in NFC.

For the INDCOL scale, we used that which was proposed by Oyserman et al. in [16]. It contained 15 items: eight for collectivism and seven for individualism. These items accounted for 88% of items across each of the 27 scales found cited in the past 20 years.

In our elicitation, we further customized each scale to suit the experimental setting by developing a translation mapping. This was necessary since it was observed by Postmes et al. [10], that in order to improve results, participants must view the whole process as relevant. The mapping process was identical for each scale: we mapped the

generic terms to specific software development terms and transformed all statements into questions. For instance, in the original *"I find that establishing a consistent routine enables me to enjoy life more"*, we replaced any incidence of the word *life* with *development*, which became *"Do you find that establishing a consistent routine enables you to enjoy development more?"* Our reasons for rewording into questions were twofold: an expert was consulted who indicated participants in the software field would be more receptive to questions, and it fit the format of other questions in the survey. The items are shown in Tables 4.1 and 4.2.

The responses to each question were recorded on a 6-point Likert scale. Typically, a high question score by an individual correlated to a high NFC or INDCOL response. During the NFC remapping process, some items had their meanings reversed *(reverse-coded)*. In the case of reverse-coded items, the applicable scores were reversed, or the reverse-coding was adjusted by reversing the points of the Likert scale. An example of a reverse-coded item would be *"I dislike unpredictable situations."* becoming *"What is your opinion of unpredictable requirements?"*. All INDCOL items that were reverse-coded in the original scale are also reverse-coded in the translation.

The NFC questions were interspersed with the INDCOL in sections corresponding to each of the three scales used (i.e. *"Strongly Dislike – Strongly Like"*, *"Not at All – Very Much"*, *"Never – Very Often"*). They were also randomly ordered within each section by the survey framework, so that no participant would be presented with them in the same order.

4.1.3 Activity-Based Protocol Elicitation

The first half-day session with the mainframe group included two consecutive hour-long observations of one subject at a time, which were video recorded and later transcribed.

Protocol analysis asks a participant to engage in some task and concurrently talk aloud. The claim is that this will result in a direct verbalization of cognitive processes [92]. The most well known of these may be the think-aloud and talk-aloud protocols. Think-aloud was originally introduced by Lewis and Rieman [93], and involves the participant thinking aloud as they perform a set of specified tasks. A related technique, talk-aloud protocol, involves participants describing their actions, but not to the degree of think-aloud where they also explain them.

Goguen and Linde [94] discuss why talk-aloud protocols are an unnatural dis-

Original	Adapted	Translation Mapping Used	Reverse-Coded
I don't like situations that are uncertain.	What is your opinion of requirements that are uncertain?	I don't like = What is your opinion of? situation = requirement	Yes (adjusted)
I dislike questions which could be answered in many different ways.	What is your opinion of coding tasks which could be implemented in many different ways?	I dislike = What is your opinion of? questions = coding tasks answered = implemented	Yes (adjusted)
I find that a well ordered life with regular hours suits my temperament.	Do you find that a well ordered development routine is helpful?	life = development suits my temperament = is helpful with regular hours = routine	
I feel uncomfortable when I don't understand the reason why an event occurred in my life.	Are you concerned when you don't understand the reason why code behaved unpredictably during development?	uncomfortable = concerned an event occurred = code behaved unpredictably	
I feel irritated when one person disagrees with what everyone else in a group believes.	Are you concerned when a team member holds a unique opinion from everyone else in the team?	irritated = concerned one person = team member disagrees = holds a unique opinion a group = the team	
I would quickly become impatient and irritated if I would not find a solution to a problem immediately.	Would you quickly become impatient or irritated if you would not find a solution to a coding problem immediately?	problem = coding problem	
I don't like to go into a situation without knowing what I can expect from it.	What is your opinion of being assigned a requirement to implement without knowing what you can expect from it?	I don't like = What is your opinion of? go into = being assigned	Yes (adjusted)
I don't like to be with people who are capable of unexpected actions.	What is your opinion of working with a team that lacks process?	I don't like = What is your opinion of? be with = work with people = team capable of unexpected actions = lacks process	Yes (adjusted)
When I have made a decision, I feel relieved.	Do you feel relieved when you have made a decision on how to implement a requirement?	made a decision = made a decision on how to implement a requirement	
I dislike it when a person's statement could mean many different things.	What is your opinion of when a team member's technical information is ambiguous?	I dislike = What is your opinion of? statement = technical information could mean many different things = is ambiguous	Yes (adjusted)
When I am confronted with a problem, I'm dying to reach a solution very quickly.	When confronted with a coding problem, do you need to reach a solution very quickly?	problem = coding problem dying = need	
I find that establishing a consistent routine enables me to enjoy life more.	Do you find that establishing a consistent routine enables you to enjoy development more?	life = development	
I enjoy having a clear and structured mode of life.	Do you enjoy having a clear and structured development process?	mode of = process life = development	
I do not usually consult many different opinions before forming my own view.	Do you consult many different opinions before forming your own view?		Yes
I dislike unpredictable situations.	What is your opinion of unpredictable requirements?	I dislike = What is your opinion of? situation = requirement	Yes (adjusted)

Table 4.1: Adapted NFC Items.

Original	**Adapted**	**Translation Mapping Used**	**Reverse-Coded**
I tend to do my own thing, and others in my family do the same.	Do you tend to work individually, and others in your team do the same?	do my own thing = work individually	
I take great pride in accomplishing what no one else can accomplish.	Are you proud of accomplishing tasks that others have not yet accomplished?	no one else can = others have not yet	
It is important to me that I perform better than others on a task.	How important to you is competitive spirit on a team?	perform better than others on a task = competitive spirit on a team	
I am unique - different from others in many respects.	Are your work habits or skills different from others on your team?	many respects = work habits or skills others = others on your team	
I like my privacy.	Do you value work privacy?	privacy = work privacy like = value	
I know my weaknesses and strengths.	Do you know your weaknesses and strengths?		
I always state my opinions very clearly.	Do you always state your opinions clearly?	*("very" removed because of scale used)*	
To understand who I am, you must see me with members of my group.	How easily can your work be understood independently from the work of others on your team?	who I am = your work	
To me, pleasure is spending time with others.	Do you enjoy working with others?	pleasure is spending time = enjoy working	Yes
I would help, within my means, if a relative was in financial difficulty.	Would you help others with their work tasks, within your means?	relative = others financial difficulty = work tasks	Yes
Before making a decision, I always consult with others.	Do you consult other team members before making a decision on how to implement a requirement?	decision = implementation decision others = team members	Yes
How I behave depends on who I am with, where I am, or both.	Do you approach problems differently depending on the team, the project or both?	behave = approach problems where = project am with = the team	Yes
I have respect for the authority figures with whom I interact.	Do you respect the work decisions made by those senior to you?	authority figures = those senior to you	Yes
I make an effort to avoid disagreements with my group members.	Do you make an effort to avoid disagreements with team members?	respect = respect the work decisions group = team	Yes
I would rather do a group paper or lab than do one alone.	Would you rather work on a task as part of a team than alone?	group paper or lab = task	Yes

Table 4.2: Adapted INDCOL Items.

Activity	Question
Call Trace: Looking at the execution trace of the program.	"What information are you looking for and why?"
Notes: Taking notes, drawing diagrams, or reading past notes.	"What information are you looking up and why?" or "What information are you recording and why?"
Consult: Either being consulted or consulting someone else.	"Why did you need to be consulted? For what particular problem?" or "Why did you need to consult that particular person? For what problem?"
Debug: Using the debugger.	"What information are you looking for?" or "What issue are you trying to resolve?"
Documentation: Looking at documentation.	"What information are you looking for?"
Reference Information: Looking at an information source.	"What information are you looking for?"
Tools: Switching between tools.	"What function does that tool provide that caused you to switch?"
Search: Using grep, in-house search tools, or searching in an editor.	"What information are you looking for?"
Pause: Not taking any action.	"What problem are you trying to understand?"
Navigation: A trail of navigating through the system.	"What information are you trying to find?"
Reaction: Recoils or says "huh?"	"What did you expect to happen?"
Focus: Leaning forward and squinting.	"What information are you looking for?" or "What are you trying to understand better?"

Table 4.3: Cue Sheet for Activity-Based Protocol.

course format, including subjects trying to provide what the experimenter desires or fluctuating between talking to oneself and to the observer. To overcome issues such as these, we used a cuing system that would ask participants specifically about when they might be forming mental visualizations or understandings. We were unable to find an already published and verified form of such an "activity-based" protocol. Therefore, we looked at daily software engineering work practices from Singer et al. [95].

They name fourteen activities observed when shadowing software engineers. Of these fourteen, six practices were applicable to our study. Applicability here refers to a work practice that could be observed, and would be of interest in the creation of tool support for program comprehension. For example, issuing a general UNIX command, or interacting with hardware, are both observable but we did not believe them to be part of program comprehension. In addition to these seven, we added five body-language and comprehension-specific activities. These include: Reference Information, Tools, Pause, Navigation, Reaction, Focus. This protocol is shown in Table 4.3.

4.1.4 Priming and Requirements Elicitation Exercise

The INDCOL results from the user profile survey showed that each group was more individualistic than collectivist (see Section 4.2.1). Following this, we prime them for

creativity to not only generate more ideas, but to also generate more creative ideas.

The priming process is as follows: first, the participants were given a set of material containing the normative manipulation and requirements elicitation exercises. Second, the participants were instructed to (a) complete the exercises in the order presented, and (b) to the best of their ability.

The normative manipulation exercises consisted of two parts. First, we primed the participants to be more individualistic—this was done by applying a series of questions, adapted to suit the context, from those used by Goncalo and Staw [17]. Second, we primed participants to be more critical by simply asking them to think critically about a particular tool they used in the course of their work. We did so since critical groups are more likely to share not only more but higher quality information [10]. Additionally, critical groups tend to avoid production blocking effects such as groupthink. The adapted individualistic manipulation and critical priming questions are shown in Table 4.4.

Original Individualistic Manipulation Question	**Adapted**
Write three statements describing yourself.	Write three statements describing your particular area(s) of expertise (i.e. topics that other team members consult you on).
Write three statements about why you think you are not like most other people.	Write three statements about how your area(s) of expertise does not overlap with that of other team members.
Write three statements about why you think it might be advantageous to stand out from other people.	Write three statements about why you think it is advantageous for individuals to have an area of expertise.
Critical Manipulation Question	**Adapted**
Instructed to be critical.	Critically comment on one particular tool, does not matter which one, which you use in the course of your work (e.g. how effective it is, or how its effectiveness could be improved). Please provide at least three comments.

Table 4.4: Normative Manipulation and Critical Priming Exercises.

The final exercise was intended to elicit requirements. The example shown is for the mainframe group. The example for the malware group is *"locating every system call to [function]"* where we used the example function *lstrcmpi*:

> *While working with assembly, there may be times when you wish you had additional tool support or visualization support.*
>
> *Please write down the tasks for which you wish you had that support. For example, note times when you think "I WISH I COULD SEE..." or "I WISH I COULD FIND..." An example might look like "locating every modification of a particular DSECT".*

Please add to the list during the day as you work. Anything you can come up with is fine, but the focus should be on problems and not solutions.

4.1.5 Nominal Group Session

The *nominal group technique* [96] can be used to improve the *quantity* and *quality* of information shared in groups, and therefore directly mitigate the detrimental effects of groupthink and production blocking. It is a structured procedure for gathering information from people that takes everyone's opinion into account. It consists of the following seven steps:

1. **Introduction:** Researchers introduce purpose of session, participants introduce themselves and area of work.
2. **Listing of Ideas:** A round table listing of ideas, which are transcribed simultaneously and projected on a screen. Talking out of turn and discussion is discouraged. New ideas that emerge are added and skipping a turn is allowed.
3. **Discussion of Ideas:** Unstructured discussion on each item to clarify, elaborate, defend or dispute items.
4. **Ranking to Select 'Top 10':** Each participant selects 10 items and orders them from 1 to 10, where 10 is the most important.
5. **Break:** Scores are tallied and items reordered by score.
6. **Discussion of Vote:** Unstructured discussion about the top 10 items – in particular, opinions on items that were either included or excluded.
7. **Re-ranking Revised 'Top 10' Items:** Each participant re-selects his or her top 10 items and assigns each a score between 0 and 100. Items can have the same score, but at least one must be given 100.

It has been shown that nominal groups produce nearly twice as many different ideas than *real groups*, and in 22 past experiments, 18 reported the performance of nominal groups to be superior to that of real groups [97]. Real groups refer to group brainstorming sessions. In regard to quality, they find that in all six studies that assessed quality, the nominal groups outperformed the real groups.

For the purpose of repeatability of our elicitation, we followed the script shown in Appendix B.

4.1.6 Exit Process

Due to constraints at the malware group's site, exit interviews were conducted at only the mainframe group with two participants. Each interview was an hour long and

was video recorded and then transcribed for analysis. These interviews were used to clarify and understand the requirements brought up during the session the day before.

An exit survey was administered to participants using the same survey platform as the user profile questionnaire. The same process was used to ensure that it was double-blind. This exit survey is used in the discussion (see Section 4.4.2) to assess how the adaptation was received and how well the normative manipulation had been followed. The exit survey questions are shown in Table 4.10.

4.2 Results of Applied Techniques

This section presents the NFC and INDCOL profiles for both the mainframe and malware groups, finding that each group has a higher than average score in both scales, with the mainframe group having a 10% higher average in both. Next, we discuss our experience with performing observational studies. Finally, we provide interesting points of discussion from the experience of using the nominal group technique. The requirements and their importance derived from this technique are discussed in Section 4.3.

4.2.1 User Profiles

The invitation to this first survey was distributed to all seven members of the mainframe group and six members of the malware group. Although the malware group consists of eight members, we were provided contact information for six since they perform reverse engineering and are therefore relevant to our study. We received a 100% response rate from the mainframe group, whereas only four members (66.6%) of the malware group responded. The difference in response rate can be explained by the fact that participation in the exercise was made mandatory by the manager in the mainframe group, whereas in the malware group, participation was voluntary.

NFC and INDCOL Profiles

In order to calculate the measures for NFC and INDCOL, reverse-coded items have had their responses corrected and the score for each individual was summed and converted to a percentage. We corrected reverse-coded responses by applying the formula $7 - response$ where response is from 1 to 6. This sum is then converted to a

percentage by dividing by the total maximum score ($6 \times 15 = 90$) and multiplying by 100. Any score that is greater than 45 (or 50%) would generally be regarded as a higher than average NFC or INDCOL scale. The formula for this conversion is:

$$\frac{\sum_{i=NFC-1}^{NFC-15} i}{6 \times 15} \times 100$$

Figure 4.1 shows the NFC and INDCOL measures for both the mainframe and malware groups. The results of the survey show that both groups have a higher than average NFC and a slightly higher INDCOL. This indicates that both groups have a greater than average need to reach solutions quickly, and are more individualistic in nature. This matched our expectations for these highly-specialized industrial software groups due to the complex and intricate nature of assembly language comprehension.

The mainframe group's median score for NFC was 71.11 (69.84 mean, 4.51 STDEV), about 20% above average, while their median score for INDCOL was 61.11 (60 mean, 3.85 STDEV), about 10% above average. The malware group's median score for NFC was 62.22 (60.56 mean, 11.98 STDEV) and for INDCOL was 51.11 median (51.39 mean, 4.09 STDEV). These results show that the mainframe group has both a considerably higher NFC and INDCOL than the malware group (approximately 10% in both cases). While the malware group does have a higher than average NFC, their INDCOL is only slightly above average.

4.2.2 Activity-Based Protocol Observations

The observation sessions were conducted by two researchers with two participants from the mainframe group. These sessions were intended to last one hour and were video recorded, and later transcribed. The first session was with a participant whose primary role was considered maintenance while the second was development. Each participant was asked to work on a typical task involving assembly language. In the first session, the participant did not have any work to complete with assembly, so he walked through a bug that he had recently fixed. Though we intended to use the activity cues, the participant still continued to think-aloud and ask us questions about what we wanted to see, though he was never told to do so.

The second participant diligently began refactoring existing assembly code, which he, himself, had written approximately 20 years ago. This developer worked so quickly that it was initially hard to follow without the context of what he was trying to

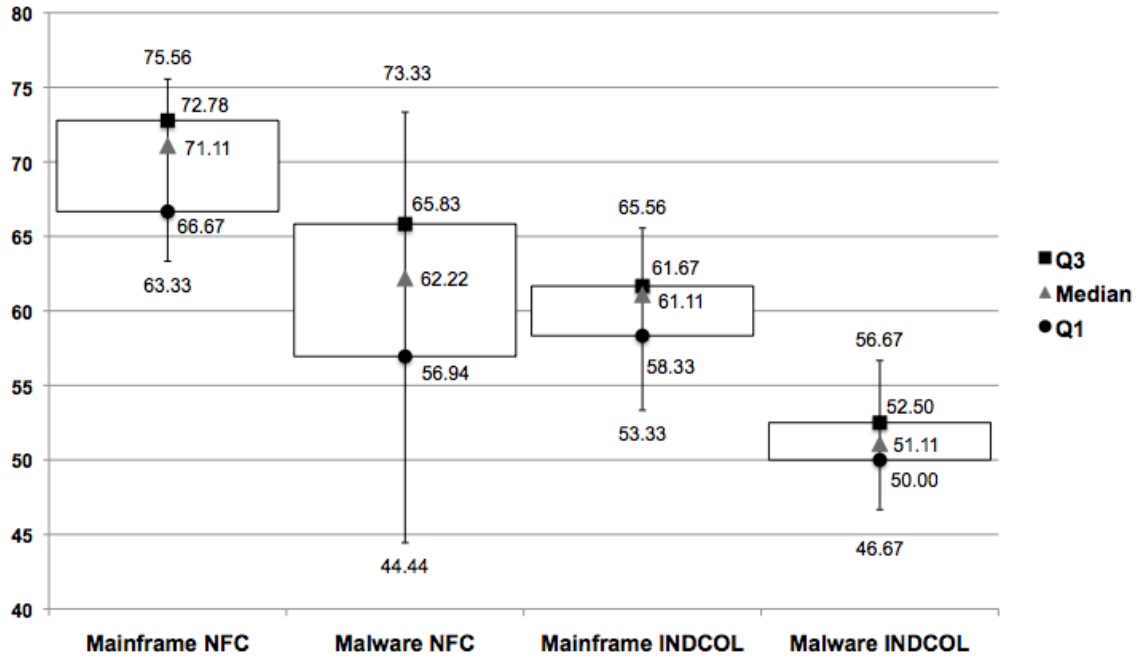

Figure 4.1: NFC and INDCOL Profiles.

accomplish. At five minutes into the session without any communication or observable activities, one researcher asked a question about what the participant was working on. After this link was established, the cuing system became much more valuable. In essence, the protocol filtered out what could have been unnecessary information in favor of data directly relevant to issues that could be assisted by visualization and analysis tool support. The process was also useful to the participant; after one particularly long pause, he remarked, *"It's funny... that pause you don't think twice about until somebody asks why you do it"*.

The two observers combined notes on issues that had been experienced by the two participants (available in Appendix C).

4.2.3 Nominal Group Session

We allotted two hours for each nominal group session. With the mainframe group, we took exactly two hours to complete the process with all seven participants and two facilitators. With the malware group, we had a total of four participants, though only three could make it to the group session and the fourth participated asynchronously by email. The session with the malware group lasted a total of 1hr 18min and was conducted by three facilitators. One facilitator was the same at each group session.

Critical Thinking and Evaluation

One participant in the mainframe group commented afterwards that it was unfortunate we had not visited earlier since he had found the sessions quite useful. He mentioned that they had intended to have regular group meetings but those meetings had never occurred.

Each group was observed to have spent the entire allotted time (10 minutes and into the following break) to assign points to issues. In the final ranking, where they could give a score between 0 and 100, many were selected at a fine-grain level (i.e. 88 versus rounding to 90).

Another aspect of each group is that they were keen to discuss issues as soon as they were brought up, and not during the given discussion period. This did not seem to affect the process, except discussion was not that plentiful during the allotted period since most of it had already occurred.

Each group had experience building tools for themselves and mentioned a desire to build solutions to the issues brought up. Within the mainframe group, this desire was more pronounced. For example, one participant commented on three issues in a row with:

"...that's actually quite, quite achievable, that we could do that... all of that stuff is feasible... we don't have it, we should do it, yes, I agree" – [another issue] – *"That's quite achievable too"* – [another issue] – *"We could do all of that stuff without much effort, we just haven't done it, we should do it".*

While we presume that participants were swayed by the rankings of others, it is not a question we asked them directly. The participant in the malware group that could not attend the group session sent his exercise sheets in advance, and one researcher filled in for him to list his issues. We then sent him the total unranked list by email and asked for his preliminary ranking. Once tallied with the group scores, we sent him the preliminary rankings to ask for his final ranking. It is interesting to note a comment that this participant wrote in his email after seeing the preliminary ranking:

"I re-ranked my answers and changed some of them as well (I am somewhat biased by the re-ordering). It has been so long since I intensively used [tool] that I forgot a few of the 'annoyances' I had..."

While we cannot provide anecdotal evidence that this is true for all participants, we do believe observable evidence exists within the differences between preliminary

and final ranking score data.

Information Sharing

Both groups appeared to share information that was previously not discussed. One participant in the mainframe group remarked to an issue *"you know you've got a fair bit of that..."* in regard to functionality they already had, and the same participant interrupted during one issue with:

"Before you go off with what you're saying, what you're saying is pretty close to what gets provided in [an existing tool we have]..."

In the malware group, after one participant had related an issue (16 - 64 bit issue), another participant pointed to the overhead and spoke in French (their primary language). The participant who mentioned the issue then said: *"Breaking news, apparently you can do that!"* The interesting observation here, is that in each group, some of the participants had worked together upwards of 10 years and yet, they did not freely discuss their issues.

4.3 Requirements Elicited

This section focuses on the requirements elicited from the nominal group sessions for the mainframe and malware group respectively. For each group, we present a table listing each requirement as well as its total preliminary and final ranking. Each requirement is described in detail and salient conversation points are additionally provided. Further, for each group, we provide the preliminary and final scores given by each individual to each requirement. We additionally use charts to show how each requirement's perceived importance changed during the nominal process. We attempt to group requirements into corresponding task areas for each group to further discuss and explore the problem space.

4.3.1 Mainframe Group: Requirements Elicited

In this section, we present the issues discussed by the mainframe group during the nominal group session. The full list is shown in Table 4.5, sorted by the final rank. With six participants, a requirement could have received the highest initial ranking of 60 and final ranking of 600. Each issue is expanded upon below based on discussion

No.	Reported Issue	Initial	Final
1.	The ability to see in context changes to any piece of source. All changes, not just the last one, including by whom, when and why.	44	440
2.	XDC debugger could be integrated with source editing (step through).	36	410
3.	A simple way to recompile all affected modules and links when I change something (similar to make).	35	400
4.	Read a DSECT, read bits and bytes, and label the various fields. IPCS functionality for a running system.	23	340
5.	Logic flow (i.e. create charts that document the code). Would need to be multi threading compliant.	19	280
6.	Better support for identifying timing problems.	11	265
7.	XDC debugger is too intrusive (interacts too much with the environment in which it is being debugged).	16	260
8.	Integration of source, documentation, logic and make.	12	260
9.	DSECT display (like XREF) that includes the fields within the DSECT.		250
10.	Looking at the assembler macro (in the source - not the listing), see from which macro library and from what level it came.	13	235
11.	Ask for a macro to expand with the argument specified in the code without assembling the whole system.	24	220
12.	Unable to use XDC (unable to add debugging traps) in the VTAM environment.	23	140
13.	Internal debugger to work with subtask engine.	15	140
14.	XDB supporting internal debug command with panel support.	9	30

Table 4.5: List of Mainframe Requirements Ordered by Final Rank.

that occurred during the group session. We use A1 through A6 to refer to each of the participants.

1. The ability to see in context changes to any piece of source (440/600 points) This requirement was the first brought up in the session, and was by A1. He mentioned being able to see in context changes to any piece of source code (not just a module or DSECT but anything). This would include all changes (not just the last one) including by whom, when and if available, why. They do currently have a basic facility to see the last change, or a specific change, but he would like to roll them all together. He mentioned at the moment that there are areas of the source code where there have been twenty changes made over the last five years by different people for different reasons all on the same piece of code and it is important to see why they were done and in context altogether.

2. XDC debugger could be integrated with source editing (410/600 points) This was part of A2's first requirement (item 7 below). He would like their debugger to be integrated with source code editing so that they have an IDE type setup where one can flip between the source and actually stepping through the code.

3. A simple way to recompile all affected modules and links (400/600 points) This was the second item brought up by A1. He wanted a simple way to recompile all affected modules and links when he changes something. A2 commented that this is similar to the *make* utility. A1 said that he would have used that term

but wanted to ensure he was speaking specifically about their environment on the mainframe. He expanded on it saying that it is not something that can be done 100% mechanically. He gave as an example, changing a comment in a common copybook which the developer knows does not require recompilation.

Similarly, he mentioned defining a new flag bit in an existing byte and that you do not need to compile anything that currently uses it unless you are making changes to them because of it. Therefore he would like the ability to find out what has been affected and adjust the list accordingly. As an example, he mentioned that you might get a list of 50 places that use this modified byte, and you can then click on the source and each one expands out to show you where it is used. You can then ignore and delete it from the list before compilation begins. He mentioned that this would save a lot of time. A2 then commented that they have previously requested this exact requirement. A3 then wanted to add to this by being able to compile the code at levels below the current level (previous versions). A2 mentioned that this can be done in Java.

4. Read a DSECT, read bits and bytes, and label the various fields (340/600 points) This was the fourth requirement from A4. He mentioned that he would like something that can read a DSECT and then read the bits and bytes and label the various fields. A1 then asked if it meant the ability to point to a piece of storage and expand it out to look like X where X is the name of something that can be used to decode the data. There was then some discussion that IPCS (Interactive Process Control Facility) currently supports something similar in a control box, however, the difference with this requirement is that it is for a running system. In other words, this would be a running system equivalent to what IPCS is doing.

5. Logic flow (i.e. create charts that document the code) (280/600 points) This was the first requirement from A4. He said that since they do not have much documentation, he would like the logic flow, as in reverse engineer and create charts that document the code. He added that it would need to understand multi-threading, which also means it would need to understand macros. He commented that they have special infrastructure macros that will not be understood by a generic tool.

6. Better support for identifying timing problems (265/600 points) This is a requirement that came out of discussion, originally brought up by A4, however, expanded upon by A1. He mentioned that at the level they work at, timing problems can be a nightmare since they are very hard to produce, let alone reproduce. It is often a thought exercise to try and look at a piece of code, or two pieces of code that interact and work out whether or not there could be timing issues. Since they deal so much with multiprocessing, they have to be very careful. He said that sometimes you can prove that there is no timing issue, but often you cannot, so you have to assume that there could be one and therefore be defensive. They often spend a lot of time tracking down problems and make a best guess that it is indeed a timing problem

that cannot be reproduced. A2 mentioned that in the worst case, they send out fixes that hopefully address the problem for the time being.

7. XDC debugger is too intrusive (260/600 points) This is the first requirement brought up by A2. He mentioned that the XDC debugger runs in the same environment as the thing that is being debugged, so by walking through it, you are changing it. He said that this is ok most of the time but in some cases it can be very difficult, and the ways around it are very messy. Coming from a virtual machine background, he found the method of debugging from outside the virtual machine far less intrusive. He did comment that it is a problem with the operating system itself and that the person who wrote the debugger could probably not have done any better at the start.

8. Integration of source, documentation, logic and make (260/600 points) This was the third requirement mentioned by A3. He would like there to be the logical integration of source code, documentation and logic. Currently if they are lucky they have documentation and if they are very lucky, they have the logic. However he would like all of them to exist on one media, integrated in their environment. He commented as well that this would require some kind of discipline among the developers that would need to be official. Further discussion mentioned that they would need management commitment in order to allow people the time for this. The consensus was that they would never have that, and that there is always a new project more important than the previous one. This means that they always need to finish the current one as soon as possible and move on.

9. DSECT display that includes the fields within the DSECT (250/600 points) This requirement was mentioned by A4 before the session officially started as something that came to mind in a recent issue he worked on. He mentioned a panel that could show all the references to various fields in a DSECT that would save him from searching through individually field by field to find the references. He mentioned XREF was useless in this regard because of the long labels. However, A1 mentioned that they are in fact there and A4 then asked how to access them and commented that he had learned something already. Everyone laughed and then A2 joked *"you could write down I wish this assembler stuff was better documented"*. He reconfirmed this issue during the session before listing his second requirement.

10. Looking at the assembler macro (in the source - not the listing), see from which macro library and from what level it came (235/600 points) This was A3's first requirement. He wished that when looking at the assembler source code, he would be able to select the macro, select a special key and get information about the macro. For example, *this* macro under *this* environment comes *this* this macro library. Also if the macro is in their own library, show information on what level it currently is. A2 mentioned that this is quite achievable.

11. Ask for a macro to expand with the argument specified in the code without assembling the whole system (220/600 points) This is A3's second requirement which followed up on the previous one. He would like to expand a macro with a specific argument without involving the entire CSECT assembly. He simply wanted to see what it looks like without assembling the whole CSECT, which may be a thousand lines long. A CSECT is an independently relocatable section of code.

A2 asked if he would be prepared to have it done in such a way that the relevant bit of the listing is shown as a window on top of the source. A3 said of course and A2 commented that this is quite achievable too. A4 then said that this would be handy since he often goes to the listing just to find the address (offset) where the various fields of a DSECT are. He said he does this especially when reading a dump, in which all you see is bytes, since the dump tool does not understand DSECTs, and the macro does not tell you what offsets the various fields are until you get to the assembly listing. A2 commented again that all this can be done without much effort and that they should simply do it.

12. Unable to use XDC in the VTAM environment (140/600 points) This was the first item brought up by A5. The main problem he had was that he wanted to use the XDC debugger within a VTAM exit environment. Specifically he wanted to put XDC debugging traps into the VTAM (production) environment. A2 commented that he mentioned something similar on his exercise sheets, which was being able to integrate the debugger with source editing so one could have an IDE setup where you can flip between the source and the debugger stepping through the code. He mentioned also that right now their debugger is too specific and gets in the way (point 7 above). P1 then commented that this actually can be done and he would show him how offline. Some private discussion ensued to clarify and P1 confirmed that it could be done.

13. Internal debugger to work with subtask engine (140/600 points) This was the third item from A4. He would like the internal debugger to work with the subtask engine. The only discussion was that he asked A1 if this were possible, which A1 confirmed.

14. XDB supporting internal debug command with panel support (30/600 points) This was the second requirement brought up by A4. Ultimately he would like a DSECT display with fields and references. He went on to explain that they have an internal debugging program (XDB) and he would like it to support internal debug commands with some panel support. For example, instead of having to look up commands (i.e. set an app trap or display the storage), that could be done within the panel. Particularly if it could read a DSECT, for example, which register the DSECT is based off of, or see a particular field.

4.3.2 Mainframe Group: Discussion of Ranking Results

While Table 4.5 shows the total for the preliminary and final rankings for each requirement, Table 4.6 further expands on this by showing the participants' individual scores. Figure 4.2 shows these as side-by-side bar charts with the lighter bar being the preliminary ranking (multiplied by 10 to use the same scale) and darker bar the final ranking. These numbers show us how individuals may have changed their scores based on the opinions of others, or perhaps how an issue was strongly felt by a particular individual and therefore they did not change their mind. We do not wish to describe these results in great detail and leave this for future work, however we will discuss some interesting observations by three different phenomena.

No.	A1		A2		A3		A4		A5		A6	
1.	5	100	9	100	8	90	8	50	8	100	6	-
2.	8	80	4	70	10	100	7	80	7	70	-	10
3.	10	100	8	70	4	80	-	50	6	50	7	50
4.	-	40	-	-	2	30	2	100	10	70	9	100
5.	7	10	2	50	5	40	3	100	1	-	8	80
6.	6	25	-	50	-	50	1	100	4	10	-	30
7.	-	-	-	100	3	90	-	-	3	70	-	-
8.	2	-	1	80	9	80	-	-	-	20	-	80
9.	3	-	3	50	1	-	4	100	-	-	10	100
10.	1	35	7	80	7	70	-	-	5	30	-	20
11.	-	20	6	70	6	90	10	20	2	20	-	-
12.	-	-	5	-	-	-	9	20	9	100	-	20
13.	9	40	-	-	-	-	6	80	-	-	-	20
14.	4	30	10	-	-	-	5	-	-	-	-	-

Table 4.6: Rankings by Mainframe Group Participant (A1 - A6).

Differing Opinions: There were of course differing opinions of requirements, some that may be based on the work process of an individual. We noticed that there is no trend towards a difference existing in a specific category (categories shown in Section 4.3.3), but we did notice some interesting differences within a requirement itself. For example, if we look at Figure 4.2 and the chart for No. 2, we see that there are varying opinions, between 100 at the maximum and 10 at the minimum. We see a similar pattern in No. 4 with half of the participants giving it a less than 50 score. Other requirements with interesting differences in opinion are 7, 9 and 12.

Individual Opinion: Similarly to above, we see a difference in score produced by a specific individual. Looking at No. 4, 5, 6 and 9, we see that A4 ranked them all at 100 (the maximum), while others ranked them far below. We also see that these requirements were those introduced by A4 so therefore he was biased towards them. It is interesting to note that while A4 did not give these requirements the best

score in his preliminary rankings, he made a push for them in his final rankings. In fact, the only score that A4 gave that was high and not his own was No. 2. We see a similar pattern with other requirements/individuals, however, the case is most present for this participant.

Shifting of Scores: A further facet to inspect is how the preliminary rankings and/or final discussion influenced individuals to change their initial opinions. In this case we are looking for a marked difference between the lighter bar and corresponding darker bar for an individual. The starkest example is in No. 7 and A2, having given it a zero ranking in preliminary, yet 100 points in the final ranking. Of further interest is that this requirement was introduced by A2. This may have been an error on A2's behalf but perhaps not. A further example can be seen in No. 8 with A6, where he went from a zero ranking to a final ranking of 80.

4.3.3 Mainframe Group: Requirement Areas and Current Work

We have classified all of the reported requirements into categories. These are shown below with their corresponding requirement numbers as ordered in Table 4.5.

- Browsing and Navigation: (0 points, observed)
- Build: 3, 11 (520 points)
- Control Flow: 6 (265 points)
- Data: 4 (340 points)
- Debugging: 7, 14 (290 points)
- De-obfuscation: (0 points, observed)
- Documentation (including comments, tags and commit messages): 5 (280 points)
- Integration (tool, system): 2, 8, 12, 13 (950 points)
- References: 9, 10 (485 points)
- Source Control: 1 (440 points)
- Source Editing: (0 points, observed)

These categories were created to the best of our ability however it is important to note that some requirements crosscut categories. For example, requirement 8 states that there should exist integration between source code, documentation, logic and make. This has been slotted into the Integration category but the requirement also mentions lack of documentation so this could also exist in the Documentation category. There are two categories that have no requirements reported from the mainframe group. These are Browsing and Navigation, and De-obfuscation. However during our activity-based protocol elicitation, we observed issues in both.

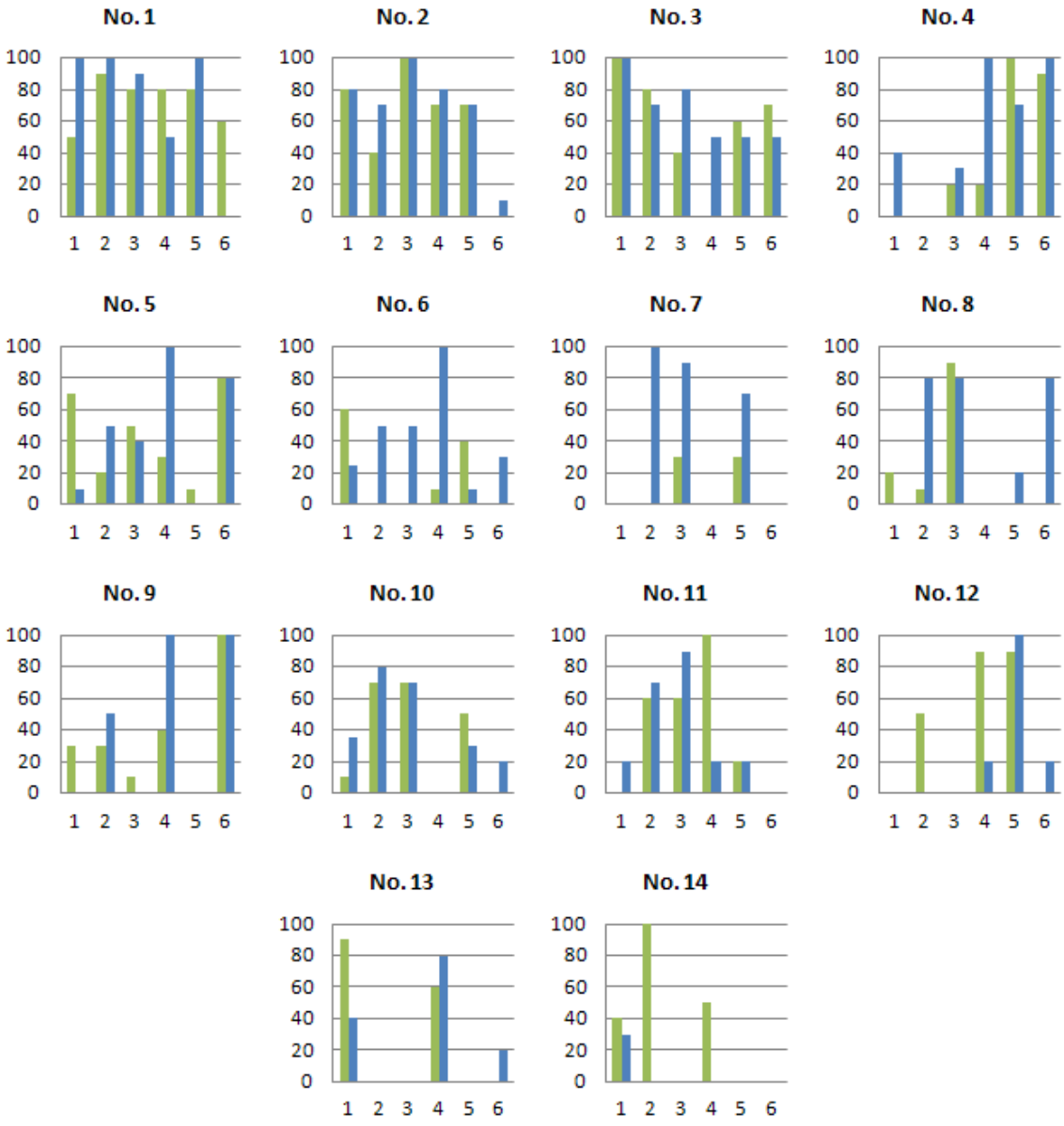

Figure 4.2: Mainframe Group Preliminary and Final Rankings.

In the case of Browsing and Navigation, we observed several issues belonging to this category (see Appendix B), for example, rudimentary bookmarking of lines of code and inefficient searching with the XREF tool. Similarly for De-obfuscation, we know that in certain mainframe systems, code has been purposely obfuscated to prevent its theft (or misuse). We know this to be true of modern software as well. However, even looking at our observation notes, we see that the mainframe group faces issues where redundant code makes the code confusing to read. While this is not done on purpose, it is a clear case of obfuscation.

Each category has the sum of its points provided. From these numbers, we see that integration of different tools and systems is the foremost issue for the mainframe group. Following this, the three most important categories are: building the system; finding references in the codebase; and performing basic source control functions.

While we aim to create solutions for both groups of assembly language developers, there are inherent challenges with creating solutions for the mainframe environment. The first being access to the systems and code themselves, and the second being that any visualization support would need to be run outside the mainframe with hooks into it. We therefore aim to create language agnostic tools with the understanding that these hooks could be created by an employee on-site. For example, we have created the Tracks [86] tool for control flow that uses its own dialect-agnostic XML format to represent function calls. Additionally, Tracks can receive messages from external sources (using sockets) to build diagrams. Although Tracks works with High-Level Assembler (HLASM) [1] code (as is used in the mainframe group), it is currently not integrated directly with the mainframe. Another possible solution to work with external tools is to use FTP to transfer log files from the mainframe, that can then be opened in another tool.

4.3.4 Malware Group: Requirements Elicited

The full list of issues for the malware group is shown in Table 4.7. With four participants, a requirement could have received the highest initial ranking of 40 and final ranking of 400. For each of the 15 issues generated, we further discuss the details for each below. One important note to mention is that IDA Pro [5] is the predominant tool for the participants so most comments are in regard to what it lacks in tool support currently. We use B1 through B4 to refer to each of the participants.

No.	Reported Issue	Initial	Final
1.	Disassemble more than one executable file at a time in IDA Pro (e.g. DLL libraries) and link between them.	31	380
2.	During dynamic analysis, changes you make (comments, renaming of functions) are lost the next time the debugger runs.	21	300
3.	When you have floating code (code that is not in an executable), it is very difficult to find the entry point so help to find possible entry points.	22	297
4.	I often get lost while going deeper in the code—where I started from or how I got there. A map of my analysis could be useful.	20	295
5.	A tagging mechanism (like in TagSea [98]) but for assembly (tag a global variable and see where it comes from, possibly with links).	27	295
6.	Easily access API documentation - e.g. a web-based doc to see how a function works.	16	293
7.	Some sort of de-obfuscation help, e.g. find and remove predicates that are always true but only there to confuse the human.	16	248
8.	In IDA Pro you do not have repeatable comments for external modules - see the comment every place you call this function.	8	225
9.	Trace replayer.	11	220
10.	Insert boilerplate documentation or comments in IDA Pro - usually tend to format comments similarly and have to do this by hand.	14	210
11.	Show the stack trace during static analysis (name or value of register pushed, instead of just the value of the stack pointer).	11	100
12.	Cross reference mechanism between a given function in an executable file to a DLL.	6	95
13.	Get a value of a constant in IDA in different formats (int, float, date)—hard to figure out what the numbers actually mean.	9	70
14.	Show possible values stored in variables in static mode, and where they came from. A linkable pop-up would be nice.	7	50
15.	IDA cannot disassemble the same file when parts are in 16 bit, 32 bit, 64 bit (have to open IDA Pro multiple times with that setting).	1	0

Table 4.7: List of Malware Requirements Ordered by Final Rank.

1. Multiple executables (380/400 points) This requirement received the highest ranking and was brought up by B3. He mentions that the problem with IDA Pro is that you can only disassemble one executable file at a time. When you want to analyze a DLL, you have to use multiple IDA Pro instances, and you have to manually make the link between the exported function used inside the executable and the library. This process is time consuming.

2. Retain dynamic analysis actions (300/400 points) This point came from B1 and was the last item on his list. During dynamic analysis, the user may make comments to the code, or also restructure it (e.g. create a function, rename, mark as data vs. code). If this happens to be a DLL that was brought in, but is no longer a part of the IDA Pro database (idb), then it is not saved and the user has to redo these actions each time; unless the user does a snapshot. However, this creates a very large file that is hard to manipulate. In essence, dynamic analysis could be persisted and replayed once you restart dynamic analysis. B2 commented that IDA Pro has a different database for dynamic information, so if the dynamic information is in the static information then it is kept, but if it is truly dynamic then it is lost.

3. Find entry points in floating code (297/400 points) This requirement was brought up by B2. Floating code refers to code that is not actually in an executable, and in such cases, it is sometimes very difficult to find the entry point. Assistance to find likely entry points into a piece of code would be helpful.

4. Map of analysis (295/400 points) This came from B4. The issue is that while the user goes deeper into the code, they often get lost, not being able to remember where they started from or how they got there. In this case, a map of the analysis would be helpful.

5. Tagging (295/400 points) B2 brought up the need to have some sort of tagging mechanism for documentation within assembly code that would be similar to TagSea for Java [98]. You could then, for example, tag a global variable and specify where it came from, possibly with links.

6. Access API documentation (293/400 points) This requirement came from B1, and was the first item on his list. This was the ability to easily access API documentation so that if there is a call to some function, then the user would be able to easily visit a web page or local help that describes how this function works. This would be easily integrated and would enumerate API documentation.

7. De-obfuscation help (248/400 points) B2 wanted some sort of de-obfuscation help, for example to find and remove predicates that are always true but only exist to confuse the analyst. Other examples of obfuscation are removing all object-oriented/functional structures to make it flat (spaghetti-like code), making arithmetic calculations more complex by adding operations that have no effect, or simply adding lots of dead code [99].

8. Cross-module repeatable comments (225/400 points) This was mentioned by B3. In IDA Pro you have repeatable comments, that is when you put a comment to a function, every place where this function is called you can see that comment in place. However, you might want to add a comment to an exported function—one that exists in another module or DLL of the executable. Then each time this exported function is called, you could see the comment in the same manner as local calls.

9. Trace replayer (220/400 points) B2 brought up that a trace replayer would be helpful. However, it was mentioned during the discussion that a trace replayer would be available in the next version of IDA Pro (6.3). At the time of this study, this was not yet available.

We later spoke with B2 about whether or not the trace replayer was as useful as he had hoped. He commented that it is a good step in the right direction, but that it

is still missing important features such as a static view of multiple trace values, the ability to start a replay from any address (replay only parts of the trace) and also recording the data in memory (not just the registers).

10. Boilerplate documentation (210/400 points) B1 mentioned the need for boilerplate documentation or comments in IDA Pro. Specifically, B1 noted *"I usually try to format all the comments similarly, which has to be done by hand. In some cases in 16 bit code, such as function calls or interrupt calls, I have to add 20 or 40 some lines"*. This would entail being able to define templates for documentation so that they could be inserted at a certain line, or at the beginning of each function.

11. Static stack trace (100/400 points) B1 also mentioned that when doing static analysis, being able to show static information from the stack would be helpful. In IDA Pro, one can see stack pointer references, but when available, another view could show possible variables which could be on the stack at this point. He mentioned *"Even if it's just the name of the register that gets pushed, or the value if it's an immediate value in the code"*.

We followed up with B1 to clarify. What he ultimately wanted is a view that shows what is on the stack when a particular address is selected. IDA can track changes fairly reliably, including taking care of calling conventions, when the code is not too obfuscated. However, in cases where the code is obfuscated, the actual push of an argument onto the stack can be several hundred lines above the call due to additional code that has no side effect. He mentioned that some obfuscation techniques could break IDA's stack pointer tracking, but when it works, the stack view would be helpful. Additionally, IDA allows the user to manually adjust the stack pointer to fix potential errors, at which point the stack view could become useful again.

12. Cross reference mechanism (95/400 points) B3 mentioned that cross-module references would be useful, such that a user could zoom between a given function, an executable part and a variable. The example which was given is a global variable which is also used in a DLL, a cross reference could be made between this global variable and the DLL. B2 commented, in order to clarify, *"matching variables from different executables or libraries"*. B1 mentioned in his preliminary ranking sheet (and gave this 10 points) that requirements #8 (cross-module repeatable comments) and #12 (cross reference mechanism) would be fixed by this.

13. Constant values in different formats (70/400 points) This was the first item brought up by B2. The idea is that if you have a value for a constant that is given in different formats, one would be able to infer what the numbers actually mean. For example, you would see a list of the possible constant values as a date, float, integer (32 bit, 64 bit) etc. This does not currently exist in IDA Pro, but exists in various hex viewers.

14. Possible static variable values (50/400 points) This was a requirement from B4. When analyzing malware in static mode, he wanted to know what values could possibly be stored in the variable, and where this value could have come from. He mentioned that a popup to show this information that could be linkable would be desirable.

15. Disassemble 16/32/64 bit parts (0/400 points) B1 mentioned that sometimes you have code with parts in 16 bit and parts in 32 bit. He said you must open the same file in different IDA Pro instances in different modes for each. Instead he wanted to be able to specify that a particular range would be 16, 32, or 64 bit code. However, after B1 listed this issue, B3 pointed to the overhead and spoke in French (their primary language). B1 then said: *"Breaking news, apparently you can do that!"* This requirement had the lowest rank of all, in all likelihood due to this fact.

4.3.5 Malware Group: Discussion of Ranking Results

As with the mainframe group, Table 4.7 shows the summary of requirements while Table 4.8 shows the participants' individual scores and Figure 4.3 graphs these same preliminary scores (lighter bars), and final scores (darker bars), by each requirement and individual. We again discuss interesting observations by the same three phenomena as the mainframe group, but do not exhaustively cover each scenario.

No.	B1		B2		B3		B4	
1.	10	100	10	100	9	100	2	80
2.	5	60	6	70	-	70	10	100
3.	-	58	7	60	6	80	9	99
4.	8	75	5	70	-	50	7	100
5.	7 (+2)[a]	95	8	100	10	100	-	-
6.	6	80	3	65	3	60	4	88
7.	4	70	9	90	-	-	3	88
8.	-	55	-	80	8	90	-	-
9.	-	55	4	85	7	-	-	80
10.	9	80	-	70	5	60	-	-
11.	1	-	-	-	2	50	8	50
12.	-	-	2	-	4	95	-	-
13.	3	-	1	-	-	-	5	70
14.	-	-	-	-	1	-	6	50
15.	-	-	-	-	-	-	1	-

Table 4.8: Rankings by Malware Group Participant (B1 - B4).

[a] B1 ranked the issue twice, only the first (7) is counted.

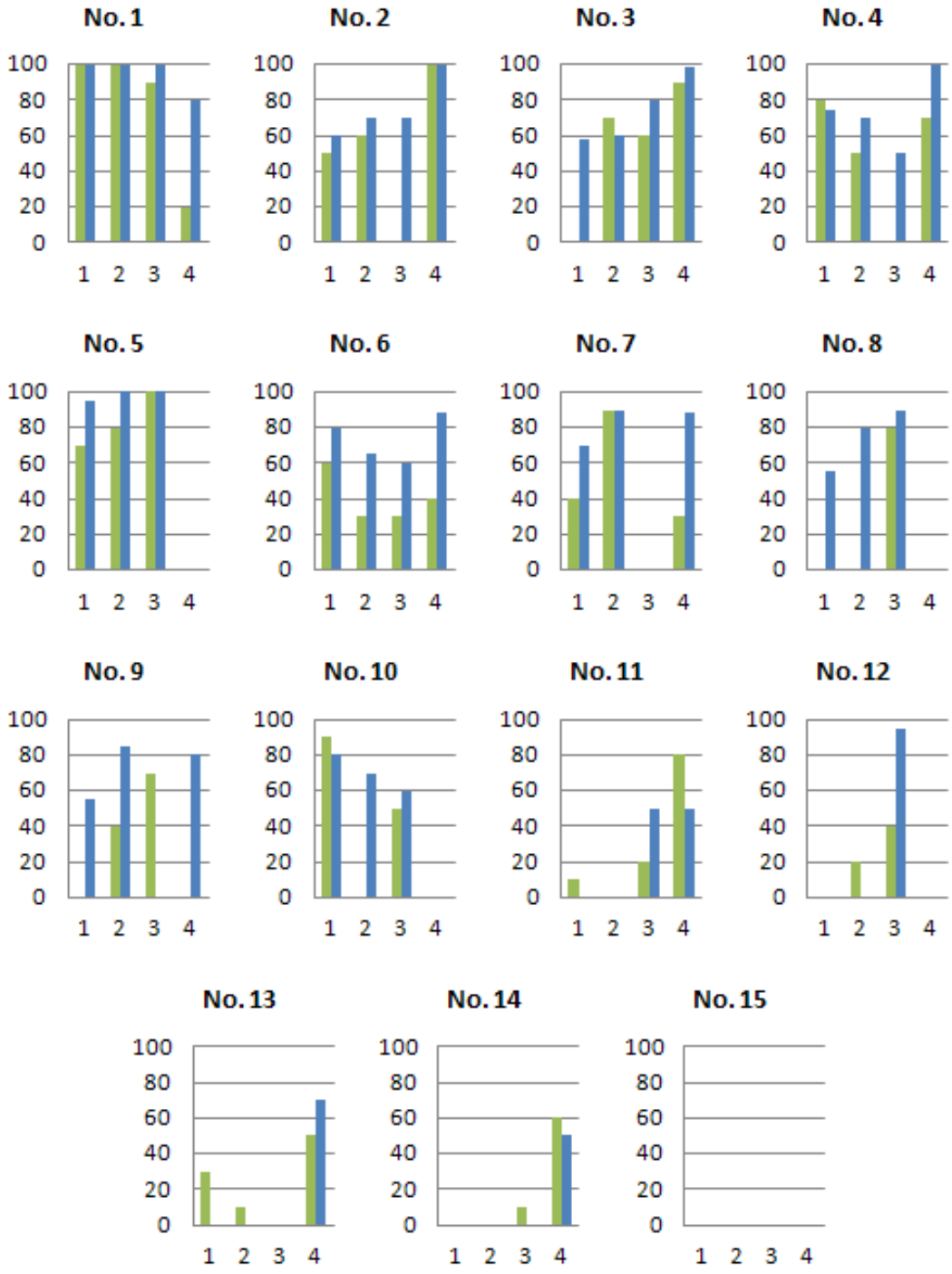

Figure 4.3: Malware Group Preliminary and Final Rankings.

Differing Opinions: Again there exists a difference of opinions between individuals on the same requirements. However, we again do not see a pattern based on category of requirement. If we look at No. 1, we see only a small difference where one individual (B4) gave it 100. Since B4 was not involved in the discussion and sent his results by email, this could have made a difference. However, No. 7, 8, 9 and 10 also show obvious differences in score not solely caused by B4.

Individual Opinion: As above, due to B4 being apart for the group session, we see that his score differs widely from others, most obviously in No. 5, 8, and 10 where he gave no score at all. However, B4 is not the only individual that displays this characteristic. The other difference we note is B3, which can easily be seen in No. 7, 9 and 12. In fact we notice an interesting pattern with No. 7 - 10 where there exists an alternation between B3 giving a high score, and B4 giving no score and vice-versa. From this we can see that B3 and B4 have contrasting opinions.

Similarly to the mainframe group, we also see the case where an individual ranks their own requirement higher than others. For example, No. 4 was only given 100 by B4 but it was also introduced by him. It could be said that since B4 was not present, he could not properly explain the necessity for this requirement to others. However this exists for B3 as well, and more notably No. 12 which has a zero ranking by everyone except for B3 who gave it 95, and it was also B3's requirement.

Shifting of Scores: Again if we look at No. 1 and B4, we see that he changed his mind considerably from his preliminary score. Since B4 was not involved in discussion, it is clear that he was strongly influenced by the scores given by others. B4 is the same individual that commented that he was reminded of his annoyances after seeing the preliminary scores given by others. However he is not the only individual who did so. There are varying differences in preliminary and final scores throughout, but we see the same for B2 in No. 9 and B3 in No. 12 (their own requirements). If we look at examples that are not introduced due to bias, we need only look at No. 8 which shows both B1 and B2 moving from a zero ranking to above 50, while this requirement was in fact introduced by B3.

4.3.6 Malware Group: Requirement Areas and Current Work

All of the above requirements are categorized into the same areas as before. They are shown below with their corresponding requirement numbers. These requirement numbers can be referenced in Table 4.7. From the summation of rankings, we see that

documentation is the largest main issue for the malware group followed by debugging issues and finding references in the codebase.

- Browsing and Navigation: 4 (295 points)
- Build: 15 (0 points, already exists)
- Control Flow: 3 (297 points)
- Data: 11, 13, 14 (220 points)
- Debugging: 2, 9 (520 points)
- De-obfuscation: 7 (248 points)
- Documentation (including comments, tags and commit messages): 5, 6, 8, 10 (1023 points)
- Integration (tool, system): (0 points)
- References: 1, 12 (475 points)
- Source Control: (0 points)
- Source Editing: (0 points)

As mentioned before, some of these categories have been included since the mainframe group had issues pertaining to them. Therefore some categories have a 0 point score for the malware group. There may also be some requirements that could potentially belong in more than one category but we have chosen what we believe to be the best fit. From the sum of points in each, we see that documentation is by far the greatest concern for the malware group. Following that, debugging and reference support are closely tied at about half the points of documentation and the other categories fall well below.

4.4 Analysis of Elicitation

In this section, we discuss the success of the social psychology techniques used for requirements elicitation within highly-specialized industrial software groups. A successful use of these techniques include the translation appearing relevant, increased information sharing through the normative manipulation techniques, and how participants reacted to their use. We investigate this through surveys and self reporting, as well as codifying statements during the nominal group sessions.

4.4.1 User Profile Survey

To address relevancy, as part of the initial user profile survey we included a final question to obtain a measure of how successful the translation mapping was in masking

the psychological background of the original questions. We asked participants *"What is your opinion on taking this survey?"*, both on a Likert Scale, and also as a free response question. On a scale of 1 (Strongly Dislike) to 6 (Strongly Like), the mean response from the mainframe group was 4.57 (above Somewhat Like) with a standard deviation of 0.53, while from the malware group, the mean response was 4.25 (slightly above Somewhat Like) with a standard deviation of 0.5. These results indicate that most participants felt that the translation mapping was relevant while only one may not have (see free responses in Table 4.9). We did not address the question of relevancy directly as we did not want our methods to be exposed, potentially biasing the subsequent requirements elicitation exercise.

Mainframe Group Responses
There are no changes I would make to the questions in the survey.
I would start with some time devoted on finding out how the team you sent the survey to works, what types of projects they work on and how the formal/informal structure of the team is. Besides, we do not work on one project at the time, we are interrupted by bugs detected, more or less urgent to solve, critical customer situations, decision changes on fly, cancelling projects in the middle due to high management changes of mind and alike.
As long as the responders are providing truthful and honest answers I believe surveys can help people understand things better.
I get the impression of questions suggesting competition within a team. In general, a team is not meant to compete within itself. A team is meant to pull together, and on the whole, within a good team, everyone respects everyone else, and all are encouraged and/or happy to ask or help others.
I found that the last question of 15 is a bit ambiguous: "How easily can your work be understood independently from the work of others on your team?"
It's hard to give accurate answers to some questions concerning time allocation. I don't keep anything but collective records e.g. how much time per week spent on customer issues (all tasks and with code in several languages).
Malware Group Responses
The survey was short and to the point.
Many of the questions are only relevant for forward software engineering and don't apply to reverse engineering at all so it was tough to answer (particularly the last 3).

Table 4.9: Free Responses on Opinion of NFC and INDCOL Survey.

4.4.2 Exit Survey

To capture results of the normative manipulation, we issued an exit survey within a week after the nominal group session. This survey determined if the participants had followed the normative manipulation instructions, whether they had found the process intrusive, and whether they found it useful. We had five participants complete the exit survey from the mainframe group and two from the malware group. Their results are summarized in Table 4.10. Any results shown with a standard deviation are averages on a Likert scale from 1 to 6 where 1 was least significant.

Asked	Reported by Mainframe Group	Reported by Malware Group
Were you able to fill out Sections A, B and C in entirety BEFORE beginning the last exercise (Section F)?	60% Yes (3/5), could not list 3 statements, did not do any ASM that day	100% Yes (2/2)
Were the objectives of the overall exercise and group session clearly explained and easy to understand?	80% Yes (4/5), not technical and detailed - more about team member's interactions than ASM coding	50% Yes (1/2), some confusion about the end goal - questions were too vague or generic
Did you feel motivated to share your personal opinions? (i.e. you said every issue that came to mind). Why or why not?	100% (5/5), discussions drew me in	100% (2/2), people involved made it easy to share, open discussion that fed into each other's ideas
Do you think that being videotaped during the process inhibited or promoted your ability to participate fully in making suggestions or discussion? (1 - Strongly Inhibited, 6 - Strongly Promoted)	4 (Slightly Promoted, 0.71 STDEV)	4 (Slightly Promoted, 0 STDEV)
Do you think that having an external person drive the discussion process inhibited or promoted your ability to participate fully in making suggestions or discussion? (1 - Strongly Inhibited, 6 - Strongly Promoted)	4.6 (between Slightly Promoted and Promoted, 0.55 STDEV)	5 (Promoted, 0 STDEV)
Was there sufficient communication during the activity? (Quantity) (1 - Very Insufficient, 6 - Very Sufficient)	4 (Slightly Sufficient, 1 STDEV)	5 (Sufficient, 0 STDEV)
Was there sufficient communication during the activity? (Quality) (1 - Very Insufficient, 6 - Very Sufficient)	4 (Slightly Sufficient, 1.1 STDEV)	5 (Sufficient, 0 STDEV)
Did you think the process was useful for the organization? If so, how?	60% Yes (3/5), communication of points otherwise not brought up, new ideas could be useful if implemented	50% Yes (1/2), 50% Unclear (1/2), confirmed issues were mostly shared with other team members, a few good ideas that might be implemented internally
Did you think the process was useful for yourself? If so, how?	80% Yes (4/5), communication of points otherwise not brought up, new ideas could be useful if implemented, made me think about my work	50% Yes (1/2), 50% Unclear (1/2), forced me to put problems I was vaguely aware of into words
Is there any way in which you think the process could be improved?	The assumption that they code assembler every day was incorrect	Future surveys should be about specific features or tools
Do you think that some part of the process should be adopted within your group? For example, this might include open discussion of issues, future work, review, etc.	60% (3/5), process improvement, more discussion on improving ASM programming environment	50% (1/2), we are already open but probably could use the process as a more formal way to discuss and evaluate issues, we do that regularly
Overall, was it a positive or negative experience?	80% Positive (4/5), 20% Neutral (1/5)	100% Positive (2/2)

Table 4.10: Exit Survey for Mainframe and Malware Groups.

From the results of the exit survey, we see that most people were able to fill out the priming exercise sheets in advance. Of the two that did not, one had begun the attempt but did not finish, while the other did not do the exercises at all. We conclude therefore, that priming was performed for the greater majority of the participants. Only one comment questioned the validity of these questions in the context of assembly language, which means they were mostly deemed relevant. In 100% of the responses, participants felt motivated to share their personal opinions. Even with such a small sample size, evidence suggests that the priming exercise was successful.

In each group, videotaping the session as well as having an external person drive the discussion was seen as promoting the participants' ability to make suggestions or engage in discussions. The quantity and quality of the communication was seen as Slightly Sufficient by the mainframe group and Sufficient by the malware group. Since the malware group has a 10% lower INDCOL score and regularly communicates about their issues, we assume they would be more communicative to begin with, accounting for this slight score difference.

We also asked about the usefulness to both the organization and to themselves. The mainframe group had just above half of its participants claiming usefulness for the organization and almost all reported it having been useful for themselves. In the malware group, one participant thought it was useful to both, and the other was unclear though he did mention, *"It might have raised a few good ideas that will be implemented internally"*. Related to being useful, we asked if there was some part of the process that should be adopted. Just over half of the participants from the mainframe group said yes, including a desire for process improvement and better communication practices. At the malware group, one participant thought that using a more formal process such as this was useful, while the other believed that they already regularly communicated about their issues. Lastly, we asked the participants if the experience was positive or negative. Not one participant answered that it was negative, and one answered neutral.

4.4.3 Analysis of Interaction

The final analysis conducted ensured that our requirements elicitation methods were successful over and above self reporting. To do so, we used Interaction Process Analysis (IPA) to codify the statements from the transcribed video sessions. IPA was developed by Bales [100] for studying small groups. It can be used to derive a set of

Social-Emotional Area	Category	Mainframe Group	Totals	Malware Group	Totals
Positive Reactions	*Shows solidarity*, raises other's status, gives help, reward	0.27%	30.65%	1.55%	28.35%
	Shows tension release, jokes, laughs, shows satisfaction	16.13%		12.89%	
	Agrees, shows passive acceptance, understands, concurs, complies	14.25%		13.92%	
Attempted Answers	*Gives suggestion*, direction, implying autonomy for other	8.33%	53.76%	8.76%	51.03%
	Gives opinion, evaluation, analysis, expresses feeling, wish	30.91%		18.56%	
	Gives orientation, information, repeats, clarifies, confirms	14.52%		23.71%	
Questions	*Asks for orientation*, information, repetition, confirmation	12.36%	15.59%	14.95%	20.62%
	Asks for opinion, evaluation, analysis, expression of feeling	2.42%		5.67%	
	Asks for suggestion, direction, possible ways of action	0.81%		0.00%	
Negative Reactions	*Disagrees*, shows passive rejection, formality, withholds help	0.00%	0.00%	0.00%	0.00%
	Shows tension, asks for help, withdraws out of field	0.00%		0.00%	
	Shows antagonism, deflated other's status, defends or asserts self	0.00%		0.00%	

Table 4.11: Interaction Process Analysis.

empirical generalizations about participant behavior. The basic process of IPA is that observers rate interactions of participants according to a scale and set of simple rules. Each response from a participant is codified against twelve categories. The twelve categories are grouped into four types of reactions: Positive Reactions, Attempted Answers, Questions and Negative Reactions. For the purposes of this study, the video sessions were transcribed and then the IPA was applied by one of the researchers. The categories are shown in Table 4.11, as well as the percentage of statements for each nominal group session. Further information on the IPA analysis can be found in [101].

In the mainframe group, the largest percentage of interaction is in *gives opinion* at 30.9% followed by *shows tension release* at 16.1%. For the malware group, the largest is in *gives orientation* followed by *gives opinion*. No interactions were codified in the final three categories (the only negative categories). Since no negative reactions were

recorded and information sharing (*attempted answers*) was ranked highest followed by *positive reactions*, we conclude that the adaptation we used made the techniques relevant, and the priming was effective.

4.4.4 Applicability to Other Groups

While we are not the first to use these research methods adopted from social psychology, we do believe we are the first to do so for software requirements elicitation in an industrial context. We report on this experience with two teams comprised of highly-specialized individuals utilizing assembly language. That is not to say that the process could not be applied successfully to groups consisting of less-specialized individuals, but we do believe there are potential issues in doing so.

We previously used the nominal group technique to elicit requirements for a web-based Community Information System (CIS) to manage First Nations' land, coastal, and marine use [102]. This session took place during a two-hour period of a two-day workshop which gathered six people who were either employed by First Nations to manage their resources, or were involved in similar projects. We could not complete user-profiling due to limited access to participants and their time. Therefore we primed only for creativity since individualistic groups generate more novel and useful ideas when told to do so, and no negative difference is observed in collectivist groups [17]. We allowed 15 minutes for silent brainstorming. The elicitation question we asked was:

> *While working as a First Nations' resource manager, there may be times when you wish you had additional tool support.*
>
> *Please write down the tasks for which you wish you had that additional support. An example might look like "easily locating every individual associated with a particular referral/proposal".*
>
> *The focus should be on the tasks and problems and not solutions.*

Priming for creativity resulted in requirements that were broad in nature, as expected for creativity primed groups. Further, while the participants had an understanding of the problem space, they did not have a specialized understanding of the technology. Therefore, many of the requirements were unrelated to the potential tool in question. For example, one participant requested *"Fieldwork and archaeology teams need weatherproof tools for mobile data collection"*.

Although the requirements generated are valuable in the context of the problem space, some of them were outside of the scope of inquiry. Additionally the sheer number of issues generated per person was much larger, which may be in large part due to the lack of scope. For six participants, 51 issues were generated; an average of 8.5 per person. In contrast, the mainframe group generated only 2.33 per person, while the malware group generated 3.75.

We posit that in terms of controlling the volume of responses, a slightly more directed elicitation question to limit the domain of interest would allow us to harness the creative inputs of the participants in a more directed manner. However, this runs the risk of stifling response creativity. In the context of this exercise, either more resources should have been allocated to allow participants full reign with their responses, or a more rigorously screened question could have been posed to direct discussion and minimise the risk of reduced creativity. In our case, we allotted two hours due to the similarly sized mainframe and malware groups. However, given the vast number of issues, we had only enough time to complete up to and including the preliminary ranking of the session.

4.4.5 Results

Based on the results of this study, we conclude that the majority of participants did not notice any irregularities in the entire process and reported it as being a positive experience. This indicates that the translation of the social psychology measures and adaptation of the normative manipulation techniques were seen as relevant. We also note that the nominal group sessions were seen as generally productive for the organization, but more so for the individual. Finally, all participants reported being motivated to share their opinions which included mentioning every one of their issues. The results of the interaction process analysis also support these claims as no negative reactions were codified, and information sharing was foremost. We therefore believe that the priming exercise was also successful.

4.5 Chapter Summary

This chapter provided the results of a novel approach to requirements elicitation. The process we followed at each site is similar, though we did perform observational sessions with the mainframe group and therefore provide requirements observed in

addition to those elicited.

We showed that each team has a higher than average Need-for-Closure (NFC) with the mainframe group having a median of 71.11 and the malware group a median of 62.22 (out of 100). This shows that both groups desired to reach solutions quickly, even if the solution they reach is incorrect. We also measured their Individualism-Collectivism (INDCOL) scale, where the mainframe group has 61.11 and the malware group has 51.11 (again out of 100). This shows that both groups are slightly above average, and are more individualistic in nature. In both scales, the mainframe group was roughly 10% higher than the malware group. Following on these results, we were able to prime the group for creativity in their individual requirements elicitation exercise.

Next, we used the nominal group session to rank and discuss the elicited requirements from each team separately. The mainframe group consisted of six members while the malware group had four. Each group generated almost the same number of requirements with fourteen for the mainframe group and fifteen for the malware group. We then categorized these requirements along with those observed within the mainframe group into eleven areas. Of these eleven, top three areas of concern for the mainframe group were: Integration, Build, and References. The malware group's top three were: Documentation, Debugging, References.

Next we looked at how scores changed over time, identifying three patterns. These were group members that had opinions in contrast with one another (Differing Opinions), one group member with a strong opinion (Individual Opinions), and discussion that may have affected an individual's score (Shifting of Scores).

Finally, we provided an analysis of our requirements elicitation process. We showed that the process was seen as both relevant, as well as useful, through self-reporting in exit surveys. We further showed that no negative responses were made during the group brainstorming session through analysis of interaction.

The elicited requirements for each group are shown in Table 4.5 and 4.7, respectively. We see that while high-level categories may be the same, the requirements themselves are specific to the group's assembly language and particular work itself. The following chapter will further examine these requirements by each category and compare and contrast them with survey results and the activity-based observations.

Chapter 5

Comparison Between Groups

The previous two chapters have reported on issues that have been experienced by two distinct groups of individuals: those responsible for development and maintenance of mainframe software systems, and those who reverse engineer malware threats. In this chapter, we will provide a summary and comparison across these groups for both the survey, as well as the requirements elicitation study. This chapter shows that while initially the problem of assembly code comprehension seemed uniform across groups, the truth is that the needs of each group are as unique as their work itself.

5.1 Comparison of Survey Results

The following section explores the comparison for survey results between the two groups: mainframe and malware. We do so by each section of the exploratory survey.

5.1.1 About the Participants

Table 5.1 shows a summary of the categories and results from each group. Additionally, it provides a column with notes on comparison between the two. In this category, there are notable differences in the preferences between the two groups. While both groups deal primarily with assembly language, it was not the preference of the malware group. While assembly language was listed as the most familiar language of all mainframe respondents, it ranked only third place within the malware group. As far as favorite programming language, assembly again was first place with the mainframe group (though not for everyone), but is not even reported as a favorite within the malware group. The final difference is with their reported favorite tools. While the

mainframe group prefers text editors, including those they can configure for their own needs, the malware group prefers IDE environments.

Asked	Mainframe Group	Malware Group	Comparison
Development Experience	88% 10+ Years	79% 10+ Years	Mainframe group has more experience in years.
Most Familiar Programming Language	100% Assembly 48% REXX 48% C/C++ 16% COBOL	93% C/C++ 67% Java 47% Assembly 27% Python	Assembly language was most familiar for the mainframe group, and third for the malware group.
Favorite Programming Language	56% Assembly 16% REXX 16% C/C++	47% C/C++ 40% Java 20% Python	Assembly language was most preferred by the mainframe group, and not at all by the malware group.
Favorite Tools	36% Text Editor 32% Debugger 16% ISPF	47% IDA Pro 40% Eclipse 33% Visual Studio 20% Text Editor	The mainframe group prefers text editors while the malware group lists primarily IDEs as their favorite tools.

Table 5.1: About the Respondents Comparison.

5.1.2 Assembly Experience

This section compares how experienced each group is with assembly, and also how they work with assembly. Table 5.2 shows the comparison between the two groups. We see that the mainframe group has reported themselves to be quite adept at both writing and understanding assembly code. There is a notable difference between their score and that of the malware group, which rated themselves lower on both but slightly more adept at understanding. Throughout, we see that this difference appears to arise because the malware group rarely writes assembly code, and rather aims to understand it. We also see that the assembly dialect that both groups use is different, with the mainframe group mostly using HLASM and the malware group using x86. As for how they use assembly language, the mainframe group is focused on development, as well as maintenance and debugging. However the malware group is more concerned with understanding malware and programs, as well as reverse engineering them. This is shown again in their most difficult and most time-consuming tasks. While the mainframe group spends the most time on testing, debugging and new systems, the malware group spends their time trying to follow control flow, data flow and dealing with malware-specific issues such as deobfuscation and decryption.

Asked	**Mainframe Group**	**Malware Group**	**Comparison**
Writing	4.42/5	2.9/5	Notable difference.
Understanding	4.46/5	3.5/5	Notable difference.
Most Proficient	80% HLASM	100% x86	Different assembly dialects used.
Used For	78% Development 30% Maintenance 13% Debugging	47% Malware understanding 33% Program understanding 20% Reverse engineering	The mainframe group performs development activities, while the malware group does analysis.
Assembly More Difficult	64% Yes 38% Many low-level operations 31% Big picture obscured 25% Knowledge of underlying hardware/OS	80% Yes 33% Many low-level operations 20% Big picture obscured 13% Translate to high-level, reliance on conventions	More malware analysts thought assembly was more difficult. However, the reasons are similar.
More Difficult	29% C/C++ 12% COBOL 8% LISP	47% No 33% Functional PLs 7% Prolog	Notable difference.
Most Difficult Task	19% Testing 19% Debugging 19% Documentation 14% Understanding others' code 10% Understanding new systems	27% Control flow 20% Data flow 13% De-obfuscation 13% Decryption	Mainframe group has development activities as their most difficult.
Most Time-Consuming Task	25% Testing 20% Debugging 20% Understanding others' code 15% Documentation 10% Understanding new systems	20% Locate behaviour 13% Control flow 13% Data flow 13% De-obfuscation 13% Decryption	Same as most difficult tasks, except for the malware group, which is locating behaviour.

Table 5.2: Assembly Experience Comparison.

Asked	Mainframe Group	Malware Group	Comparison
Primary Tool	68% Text editor 12% HLASM Assembler	87% IDA Pro	Mainframe group's text editor versus IDA Pro disassembler.
Secondary Tool	60% Debugger	33% Hex editors 27% WinDbg 20% IDA Pro plugins	Debugger is a common secondary tool.
Deficiencies	38% None 21% Text features (syntax highlighting, syntax checking) 13% Navigation within code	20% Lack of integration 13% Instruction assistance 13% Documentation (notes) 13% Conversion to high-level	Some mainframe respondents mention syntax features and navigation. The malware group needs integration between their existing tools.
Best Features	31% Data (register/var contents, memory/data flow) 19% Single step execution 13% Syntax highlighting 13% Trace or dump output	20% IDA Pro graph view 13% IDA Pro extensibility 13% IDA Pro search patterns 13% Inspect and modify heap, registers, stack	Malware group's best features are IDA Pro specific.

Table 5.3: Current Tool Use Comparison.

The one area where the two groups had common ground is in the reasons why they believed assembly language was in fact more difficult than other languages. The reasons they listed are almost the same for both groups. These reasons included that there are so many low-level operations that the big picture of what the code is doing is obscured.

5.1.3 Current Tools

This section looks at the current tools of each group. We see in Table 5.3, that the mainframe group primarily uses a text editor and a debugger, while the malware group relies heavily on IDA Pro. As far as deficiencies, the mainframe group mentions syntax-related issues as well as code navigation. The malware group's largest issue is that there is no integration between the tools they use, but also that they need assistance with instruction definitions, being able to take notes, as well as converting to C. Therefore, these deficiencies have no common ground. We see that this is true

Asked	Mainframe Group	Malware Group	Comparison
Beacons	76% Specific instructions 16% Comments 16% Macros 16% Loops 16% None	27% Function calls (control flow) 27% Data usage 27% Coding conventions 27% Function definitions	Notable difference of opinion.
Task-Focused UI	36% No 32% Yes 32% Unsure	100% Yes	Notable difference.
Zoom By	29% Subroutines 15% Do not have long modules (N/A) 12% Macros 12% CSECTs	40% Functions 20% Modules	Functions (or subroutines) are common.
Additional	33%Following links (branches, cross-refs, declarations)		

Table 5.4: Browsing and Navigation Comparison.

for the best features of their tools as well. While the malware group's best features include particular functionality of IDA Pro, the mainframe group is most interested in features pertaining to data. While this comes up for the malware group as well, it is much lower on the list.

5.1.4 Browsing and Navigation

As far as beacons in the code, there was no similarity between responses. Table 5.4 shows that while the mainframe group was concerned with specific instructions, the malware group was more concerned with other issues such as control flow. In fact, macros came up often as an issue for the mainframe group, which was not an issue at all for the malware group. The task-focused UI was of high interest within the malware group, but created a divide within the mainframe group. We do however see some commonality with a question of how code could be zoomed into, where both groups mentioned subroutines (or functions). Only the mainframe group gave a response to additional issues within this section.

5.1.5 Debugging

Table 5.5 shows results of debugging features on a scale from 1 to 5. We see that generally the values are quite similar. The top three are the same for both groups,

Asked	Mainframe Group	Malware Group	Comparison
What are the arguments and results of a function?	4.76	4.53	*Mainframe group's most important.
How does control flow reach a particular location?	4.68	4.60	*Malware group's most important.
Where is a particular variable set, used or queried?	4.6	4.60	*Malware group's most important.
Where is a particular variable declared?	3.76	3.53	*Least important for all.
Where is a particular data object accessed?	4.28	4.33	
What are the inputs and outputs of a module?	4.44	4.13	
Features Missing	Data flow concerns	Varied	Data flow is an issue for both groups.
Debugger Mockup	Positive	Negative	Notable difference.

Table 5.5: Debugging Comparison.

as are the bottom three. Additionally both groups agree that *where a variable is declared* was least important. However, the most important for mainframe respondents was *the arguments and results of a function*, whereas for the malware group, we see that *control flow* and *where a variable is set/used/queried* is most important. The difference in scores may be negligible, however, we do see a trend that puts control flow ahead of other concerns for the malware group.

With regard to features missing, the mainframe group had only data flow concerns. The malware group did not converge but had a multitude of responses, including: *data flow, trace diffs, memory view, stepping backward, re-running system with reg/var values, access list to specific memory addresses, simulating execution statically, multi-application debugging, standard so all tools can communicate.* We see that data flow was an issue for both groups.

Finally, the mockup tool (Figure 3.1) discussed during interviews showed promise with the mainframe developers, but not with the malware analysts, so this type of tool will most likely not be helpful for both.

5.1.6 Control Flow

We see that control flow was a larger concern for the malware group. On a scale of 1 - 7, both forward and reversed control flow was more important for them by a factor of greater than one. We also see that they had further static concerns. However as far

as dynamic concerns, both groups were interested in how often paths are executed, though the malware group was also interested in being able to compare traces and have multi-threaded trace support. As far as data that could be mined, the views from the two groups did not share any similarity. The mainframe group was interested in data, as well as performance. However, the malware group was more interested in call patterns, as well as comparing traces and seeing how control flow reached a specific point. Table 5.6 summarizes these differences.

Asked	**Mainframe Group**	**Malware Group**	**Comparison**
Static Concerns	None	20% Loops and recursion	
Dynamic Concerns	8% Most executed paths	7% Multi-threaded 7% Compare traces 7% Branch frequency	Path frequency is an issue for both groups.
Forward CF	5.0/7 Useful	6.38/7 Useful	More useful for malware group.
Reversed CF	5.08/7 Useful 90% Useful	6.15/7 Useful 87% Useful	More useful for malware group.
Data to Mine	24% Register values/mapping and memory usage (data) 12% Performance 12% System, subroutine call statistics	47% Call patterns 13% Compare traces 12% How to reach execution points (jump conds)	Notable difference.

Table 5.6: Control Flow Comparison.

5.1.7 Potential Tools

Table 5.7 shows the comparison for this section on ideas for potential tools. First, we asked about features common to IDEs. We reorder these elements from the original asking order to the order of most important to least. We see that in fact both groups have the same order of importance, though each item rated lower in the mainframe group than for the malware group.

As for proof of concept tools, we see that LegaSee may be useful in the mainframe context, for which it was originally designed. However, we see that there was effectively no perceived usefulness from the malware group. When we spoke with the malware analysts about this, they told us since there will usually be one very large module, combining this view with a treemap will be the most effective. The engineer can then move from large granularity to small granularity.

Asked	**Mainframe Group**	**Malware Group**	**Comparison**
Search for References Go to Declaration Syntax Highlighting Syntax Checking Code Completion	3.96 3.88 3.52 3.48 2.56	4.73 4.60 4.40 3.33 2.93	Same order for both groups.
LegaSee	40% Useful 12% Not Useful	0% Useful	May be useful for the mainframe group.
MapUI	31% Useful	33% Useful 20% Unsure	May be useful for both groups.
High-Level Split View	28% Not useful. Confusion of using assembler to develop in high-level PL.	85% Useful. Already exists in VS Studio, and Hex-Rays decompiler plugin.	Only useful for the malware group.
UML Diagrams	7% state 7% class	33% state 13% activity 7% package	More positive response from the malware group.
Wish List	- Pattern recognition. - Better macro processor, language and visualization. - Better debugging and breakpoints. - Better profiler to improve performance.	- Better integration with other tools. - Meta-assembly to push back and forth with other tools. - Data flow, sequence viewer, pattern recognition, documentation, creating C from ASM to guess what a function is doing, omniscient debugging.	

Table 5.7: Potential Tool Comparison.

For the MapUI tool, we see that there is a lukewarm response to its usefulness from both groups. The high-level split view was a popular idea for malware analysts, however, it was either confusing or not useful to the mainframe group.

UML diagrams showed more promise within the malware group, especially for state diagrams. Since state is mentioned for both, there may be some potential in this area, even though it was only mentioned by a small number of respondents.

The wish list varies greatly between groups. We see again, in the mainframe group, the need for better pattern recognition and macro support, and better debugging and performance. Within the malware group, we see again that better integration between tools is needed, as well as data flow and control flow support, better documentation capabilities and malware-safe debugging.

5.1.8 Comparison Summary

The comparison of groups in this section has shown that both groups not only work with different assembly dialects, but also with completely different purposes in mind. The mainframe group for developing and maintaining existing systems written in HLASM, versus the malware group, which tries to understand disassembled x86 code. We also see that the mainframe group's toolset is limited to text editors, whereas the malware group primarily uses IDA Pro, and therefore needs other tools to integrate with it.

Further, we see that the profiles of the developers is different. We note that while assembly is the more familiar *and* the most preferred language for mainframe developers, it is much less familiar and not even a favorite of malware analysts. The malware analysts also prefer IDE environments which mainframe developers do not.

While differences exist, there are similarities as well. They do agree on why assembly language is more difficult: there are simply too many low-level operations that obscure the big picture. Unfortunately, while the underlying reason behind each group's difficulties may be the same, the solution to ease these difficulties is not. Table 5.8 shows the highest-rated issues that define the problem space for each group.

5.2 Comparison of Requirements Elicited

While both groups use assembly language and face shortcomings in their current tool support, we can see from the elicited requirements that assembly language in itself is

Area	Mainframe Group	Malware Group
Existing Tool Support	Syntax (highlighting, checking) Navigation within code	Better tool integration Instruction definitions Documentation support Higher-level representation
Browsing and Navigation	Specific instructions Links (branches, xrefs, decls) Comments Macros Loops	Function calls Coding conventions Data usage Function definitions Modules
Debugging	Data flow	Save debugging state Reset values/rerun Data flow Static simulation Multi-threaded support in IDA Pro
Control Flow	Register values/mapping Memory usage Performance System/subroutine call statistics Most executed paths	System call patterns Loops and recursion Compare traces Data to reach execution point Branch frequency Multi-threaded traces
Potential Tools	LegaSee	High-Level Split View UML Diagrams (State, Activity)

Table 5.8: Comparison of Features from Exploratory Survey.

not a great enough similarity to enable the creation of the same tools for both groups. These groups are different in many ways. The mainframe group uses HLASM code to write and maintain mainframe systems, whereas the malware group reverse engineers x86 code to analyze security threats, all the while using completely different tool sets to accomplish these tasks. However, while these groups are very different, there are a few similarities in the issues they face. Both their differences and similarities are discussed below by requirement category.

Browsing and Navigation As discussed previously, while this issue came up explicitly in only the malware group's session, we have been able to determine through observation that issues in this area during our activity-based observation sessions (Appendix B). The malware group's requirement was a map of their analysis to save where they have gone so that they do not get lost. There was no similar observation at the mainframe group, however, we did note that their search functionality only found one item at a time, and that they required many screens open at once with different segments of the same module, since it is so difficult to navigate.

Build This only came up as an issue for the mainframe group as their primary work involves development. It is important to note that executing malware requires a specialized environment. However, issues related to this were not considered part of this category.

Control Flow One of the similarities that is not obvious in the elicited requirements is the need for control flow tools. While the mainframe group mentioned the need for a tool to document the code (logic flow), this was not brought up by the malware group. However this was the first requirement ever brought up by the malware group at the beginning of the AVA project and was created at their request. While this may not have been brought up in the session, it was indeed a high priority issue for the malware group. The other issue we have categorized as control flow is similar for both the mainframe and malware group. From our observations at the mainframe group, we find that it is difficult to find the main task, whereas the malware group brought up in the group session that it can be difficult to find the entry point.

Data The mainframe group's data requirement involved reading a DSECT and expanding it out by decoding the data. The malware group had two data requirements. Their first was to show possible values of a constant (*int*, *float*, *date*) so that the analyst could make a best guess at what the data contains. Their second was to statically show possible values stored in a variable and where these assignments could have occurred.

Debugging This was one area in which there were many issues brought up by both the mainframe and malware groups. The mainframe group had many issues related to debugging that have been categorized into more specific areas, though they still fall under the debugging umbrella. For example, the integration of the XDC tool in the VTAM environment was instead placed under Integration. While both groups had many issues with debugging (including those observed at mainframe), they were specific to their own tooling environments.

De-obfuscation This issue came up explicitly for the malware group, and not for the mainframe group. However, as we noted previously, this is an issue we came across during our observations at the mainframe group in the case of redundant code, making the code difficult to read.

Documentation Both the mainframe and malware groups had a number of issues in regards to documentation. The mainframe group mentioned two during their session which were *charts that document the code*, as well as *integration of their documentation artifacts*, neither of which have counterparts in the malware group. However, the malware group brought up four issues that do have comparisons to mainframe from the observation sessions. The first is a tagging mechanism and in the mainframe group we observed that they use single character bookmarking, or comments, for lines of code as well as TODOs, which is insufficient. The malware group's second documentation requirement was to easily access API documentation. We see for the mainframe group that they also reference IBM manuals and have to do so manually. Two of the four from the malware group brought up shortcomings with the comment system in IDA Pro. At the mainframe group, we saw that multiple developers printed off the code and wrote comments directly on the paper, or used sticky notes to make comments. The issues for this category are therefore closely comparable.

Integration This issue only came up for the mainframe group in the group session. There are many tools/environments that they use within the mainframe but there were many issues listed about them not working together. We know that the malware group predominantly uses IDA Pro although does occasionally use other tools (such as *PaiMei* [21]). In the survey, they mentioned lack of integration as the largest deficiency with IDA Pro.

References The mainframe group's reference requirements are for references to fields within a DSECT, as well as seeing which library (and level), an assembler macro came from. The malware group needed a link between the main executable file and DLL's that they referenced. The observation we noted for the mainframe group was dependencies between code and modules. This observation is quite similar to the malware group's needs.

Source Control This issue only arose for the mainframe group as their work differs from the malware group as development-based.

5.2.1 Comparison Summary

While there may be similarity across the two groups and the issues they face, we see that both groups reported issues related directly to their own work processes rather than issues with assembly language in general. For example, the mainframe group has issues related to version control that most high-level systems have already addressed. This is a reflection of antiquated mainframe tools rather than an issue with assembly language itself. Even where similarities do exist, the same tool support could likely not be built to satisfy them. An interesting note is that we found more similarity between the mainframe and malware groups from our activity-based elicitation than what was self-reported by the mainframe group.

Table 5.9 shows the number of issues observed at the mainframe group and reported by the mainframe and malware groups. Darker shaded rows show categories that were self-reported by both groups. Lighter shaded rows show categories that had issues reported by the malware groups but only observed at the mainframe group and not reported. Unshaded rows show categories that existed for either group, but not both. An additional twelfth category was introduced based on our observations which is *source editing*. The total number of issues we observed across both hour-long sessions was 24, with the majority belonging to *browsing and navigation* followed closely by *debugging* and *documentation*.

Category	**Observed at Mainframe Group**	**Reported by Mainframe Group**	**Reported by Malware Group**
Browsing and Navigation	Yes (5 Issues)	No (0 points, 0 Issues)	Yes (295 points, 1 Issue)
Build	Yes (3 Issues)	Yes (520 points, 2 Issues)	No (0 points, 1 Issue)
Control Flow	Yes (2 Issues)	Yes (265 points, 1 Issue)	Yes (297 points, 1 Issue)
Data	No (0 Issues)	Yes (340 points, 1 Issue)	Yes (220 points, 3 Issues)
Debugging	Yes (4 Issues)	Yes (290 points, 2 Issues)	Yes (520 points, 2 Issues)
De-obfuscation	Yes (1 Issue)	No (0 points, 0 Issues)	Yes (248 points, 1 Issue)
Documentation	Yes (4 Issues)	Yes (280 points, 1 Issue)	Yes (1023 points, 4 Issues)
Integration	No (0 Issues)	Yes (950 points, 4 Issues)	No (0 points, 0 Issues)
References	Yes 1 Issue	Yes (485 points, 2 Issues)	Yes (475 points, 2 Issues)
Source Control	Yes (1 Issue)	Yes (440 Points, 1 Issue)	No (0 points, 0 Issues)
Source Editing	Yes (3 Issues)	No (0 Issues, 0 Issues)	No (0 Issues, 0 Issues)

Table 5.9: Comparison of Issues for Mainframe and Malware Groups[a].

[a]Dark rows were self-reported by both the mainframe and malware groups, lighter rows show issues reported by the malware group but only observed at the mainframe group, unshaded rows existed for either the mainframe of malware group, but not both.

5.3 Ships Passing in the Night?

We have shown that while there is crossover in specific instances with both groups, there is no one tool framework to rule them all. The reason for this is twofold. First, the assembly language used is quite technically different. Second, the type of work that each group does is highly-specialized. The initial goal of the project was to create one set of assembly tools that would help with the blanket topic of assembly visualization and analysis. However, we have shown that while tools would definitely help both groups, our results have clearly established fundamental disparities that would indicate no one set of tools would simultaneously be equally effective for both groups. In the most stark example, we see that while 40% of mainframe developers said a tool, such as a high-level construct visualiser, could be useful, 0% of malware analysts said it would be. Conversely while control flow issues seem to be a bigger concern for the malware analysts, it still remains a large concern for the mainframe developers as well.

We therefore conclude that while some tools could be built to work with both groups, we first must ask if the tool is useful, and second we must procure the data to make its use possible. This can be a large challenge, especially in mainframe systems. These challenges are also further discussed in the following chapter. The most important factor to consider may be that of flexibility. A tool capable of visualizing whatever is passed to it, through a flexible intermediate language or model, is necessary. However, this tool must also be extensible for purposes specific to the context it is applied to.

In concluding this section, we bring together the issues faced by both groups from the surveys, as well as the requirements elicitation, in one place. Table 5.10 combines the issues identified through the survey, with those from the requirements elicitation. They are listed by category identified through the requirements elicitation. Finally, issues that are present within both groups are highlighted by using a boldfaced font.

5.4 Chapter Summary

This chapter provided a direct comparison between the mainframe developers and malware analysts by each section of the survey presented in Chapter 3. We then provided a comparison of the two groups by each category identified in the requirements elicitation in Chapter 4. Each section provided a comparison table that outlines

Category	Issues for the Mainframe Group	Issues for the Malware Group
Browsing and Navigation	Navigation within code Specific instructions Links, Comments, Macros, Loops Observed: search finds one at a time Observed: many screens open at once to view same module	Function calls Coding conventions Data usage Function definitions Modules Map of analysis
Build	Recompile affected modules/links (make) Macro expansion given an argument without assembling entire system Observed: register usage Observed: stub errors Observed: compile errors to find changes	Disassemble file with parts in 16, 32, 64 bit (already exists)
Control Flow	Register values/mapping Memory usage Performance **System/subroutine call statistics** **Most executed paths** **Identifying timing problems** Observed: **difficult to find main task** Observed: **Multi-threaded traces**	**System call patterns** Loops and recursion Compare traces Data to reach execution point **Branch frequency** **Multi-threaded traces** **Find entry point in floating code**
Data	Visualiser Label various fields in a DSECT	Stack trace during static analysis Value of constant in different formats Possible values stored in variables, and where they came from, in static mode
Debugging	**Data flow** XDC debugger is too intrusive XDB supporting internal debug command with panel Observed: No breakpoints in XDC Observed: need to trap events Observed: correct location to debug	Save debugging state Reset values/rerun **Data flow** Static simulation Multi-threaded support for IDA Pro Trace replayer
De-obfuscation	Observed: redundant code	De-obfuscation help (remove code only there to confuse human)
Documentation	**Logic flow (charts that document the code)** Observed: **characters and comments used to bookmark** Observed: **Reference IBM manuals** Observed: print code to write comments	Instruction definitions Documentation support Higher-level representation **UML Diagrams (State, Activity)** **Tagging mechanism** **Access API documentation easily** Repeatable comments for external modules Insert boilerplate documentation/comments
Integration	XDC debugger integrated with source editing Integrate source, documentation, logic and make Use XDC in VTAM environment Internal debugger work with subtask engine	Better tool integration High-Level Split View
References	DSECT field display See assembler macro library/level in source Observed: **dependencies between code and modules**	Disassemble multiple files at once, and link between them **Cross-reference between function and DLL**
Source Control	In context changes to any piece of source (who by and why) Observed: module replacement issues	
Source Editing	Syntax Highlighting Syntax checking Observed: No save reminder Observed: Code templates	

Table 5.10: Summary of Issues from Survey and Requirements Elicitation[a].

[a]Similar issues present in both groups are boldfaced.

where issues lie for each group to further highlight differences and/or commonality between them. The final section of the chapter discussed that, while similarities do exist, ultimately no one set of tools can equally satisfy the spectrum which our two stakeholders establish in terms of program comprehension. While this may not correspond directly to the initial project plan, it does not mean to say that tools cannot be built with multiple purposes in mind.

This concludes Phase II of the study, where we have established from the surveys and requirements elicitation that there are fundamental disparities between the needs of these stakeholders. The research questions that Chapters 3, 4 and 5 have answered are: *What are the requirements currently not being met in the comprehension of assembly code within two unique groups: mainframe developers and malware analysts?* and *What are the similarities and differences in the requirements?* However, in order to more concretely determine how these disparities may play out in a framework, Phase III explores proof of concept tools aligned with these requirements and applied to stakeholder specific tasks. In our investigation of higher-level tool support in this low-level domain, we have created one tool in particular that has been shown to be useful for both stakeholder groups.

Chapter 6

Design and Implementation

This chapter discusses the design and implementation of proof of concept tool support built alongside survey feedback and elicited requirements described in previous chapters. We first introduce the architecture of the AVA framework[1]. Next, we introduce how we obtain data for both the mainframe and malware groups, as well as the data formats used in each tool. Following that, we discuss each tool belonging to the framework. First we introduce our sequence diagramming tool, called *Tracks*, which provides flexible visualizations for control flow. Next we show the visualiser extension called *LegaSee*, which shows a high-level view of specific constructs within a mainframe system. Finally, we show *REwind*, a debugging tool that can save and rerun a user's reverse engineering actions through a state diagram. At the end of this chapter, we discuss tools that our colleagues have built to satisfy malware requirements identified in the work of this dissertation.

6.1 AVA Framework Overview

The AVA framework consists of the program comprehension tools, as well as the integration with other tools through further plugin and communication mechanisms. Additionally, the tool suite can work directly off of text files. Figure 6.1 shows an overview of the AVA framework. All of the user-interface level tools are built as plugins using the Eclipse Rich-Client-Platform (RCP). We decided to use Eclipse due to the number of modeling plugins and tools already available as open source for the Eclipse platform, but also familiarity of Eclipse within the malware group's user base

[1] Available at: `https://github.com/jebaldwin/AVA`

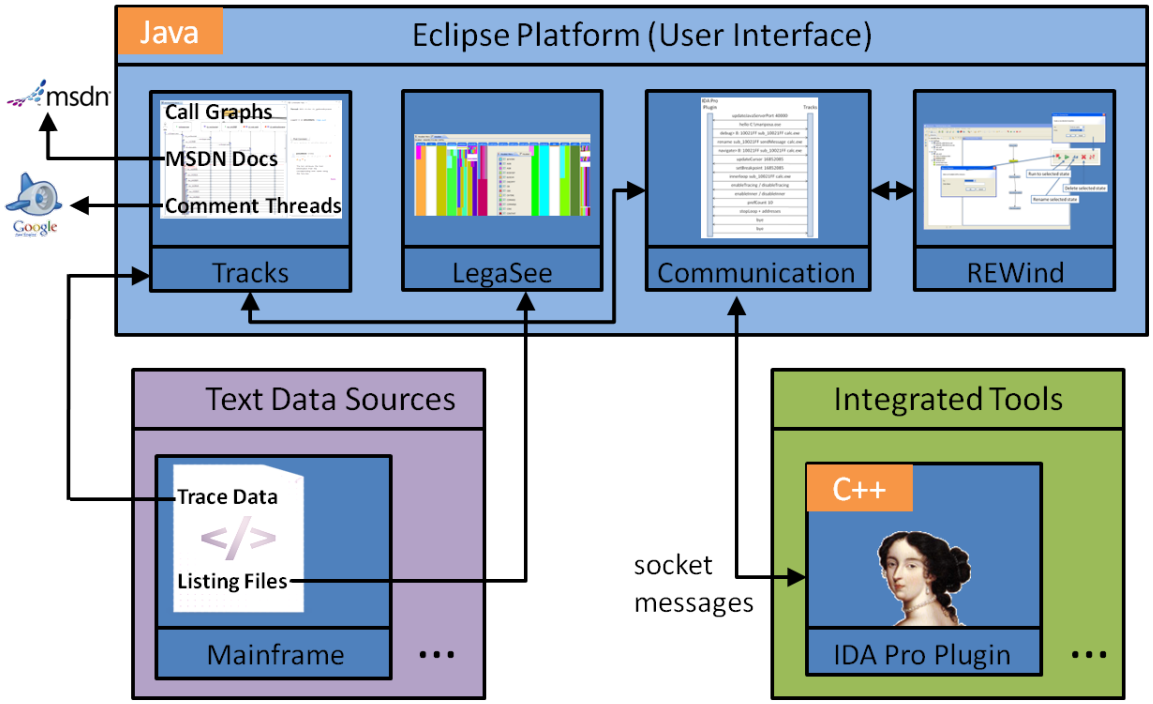

Figure 6.1: AVA Framework Architecture Overview.

as well as research groups such as CHISEL.

While we could build these tools within Eclipse, we still needed to integrate our tools with those already in use. Our current implementation is built to work directly with IDA Pro through its own plugin architecture. Since our IDA Pro plugins are written in C++ and our user-interface written in Java, we provide a communication module within Eclipse that is explained later in Section 6.1.3.

Finally, even though we showed that integration is possible with the predominant tool used by malware analysts, we still face limitations as far as integration with the mainframe environment. These limitations are discussed further in Section 7.1.1. To summarize, it was difficult to gain access to mainframe systems, in addition to the steep learning curve required to program such a system. We show that the communication structure we use is flexible enough to allow integration with multiple other systems but in the meantime, rely on text output from the mainframe, which is manually fed into our suite of tools.

6.1.1 AVA User-Interface (Eclipse)

AVA consists of either a set of Eclipse plugins, or a single RCP application, depending on user preference. The Eclipse plugins (shown in Figure 6.1) include: Tracks, which includes sequence diagrams, links to MSDN documentation as well as collaboration and documentation mechanisms; LegaSee, which provides a 'bars-and-stripes' style representation of code constructs; and REwind, a state diagram tool for saving and re-running a user's debugging actions. AVA also includes the communication module that can be used across all tools. The RCP application is provided as a simplified Eclipse executable containing these tools and no others. The intention was that some users may not be familiar with Eclipse and providing the entire Eclipse IDE along with AVA would overcomplicate the AVA toolset. These tools are discussed separately in detail later in this chapter.

While the malware group may have been familiar and even fond of Eclipse as shown in the surveys, they also use IDA Pro predominantly in their analysis activities. Therefore, it was imperative that our tools worked seamlessly alongside IDA Pro. This gave us the opportunity to leverage the IDA Pro plugin framework to create this link, as well as retrieve the data necessary for our tools. We realize that all tools built as part of the AVA framework will need to be flexible in order to be used alongside other systems, such as mainframes. The communication mechanism implemented to this end is discussed in Section 6.1.3.

6.1.2 IDA Pro Plugin

The IDA Pro plugin was written in C++ and supports communication between IDA Pro and other tools. The plugin is also responsible for providing the data necessary for the AVA tool suite, including static control flow information and real-time information such as dynamic control flow, and user interactions with the system.

The IDA Pro plugin framework [103] provides event notifications for four different types of events. These include HT_IDP for processor module events, HT_IDB for database events, HT_UI for user interface events and finally HT_DBG for debugger events. To use these events, we simply register a callback function that hooks to the notification point. With these callback functions, we can retrieve data related to the event type, including the notification code. For example, notification codes for the HT_DBG event could include dbg_process_start for the process having started, dbg_bpt for a user breakpoint being hit, or even dbg_trace for an instruction being

executed. The latter requires that step tracing be enabled which can also be set through the plugin. Using these events, we can send messages with the pertaining information to AVA to be processed.

Additionally the IDA Pro plugin receives messages from AVA. One common event we may see is to update the cursor to a specific location. Should we receive this message, we can use the *jumpto* method within the IDA Pro plugin to move the user's cursor to the address provided as an argument.

Further information regarding these events and how the IDA Pro plugin framework can be used is found in the manual written by Steve Micallef [103].

6.1.3 Communication Mechanism Module

To maintain independence between tools, we implemented a messaging mechanism for AVA such that any external tool could be extended to work with the AVA framework. We discuss this messaging in regard to IDA Pro for which it was implemented. However, it is important to note that this communication is not limited to IDA Pro.

To pass messages between IDA Pro and the AVA framework, we needed to create the IDA Pro plugin, discussed above, to generate the required data. We then send these messages using sockets since IDA Pro plugins are written in C++ and AVA is written in Java. Figure 6.2 shows a sample of messages that can be exchanged between IDA Pro and the Tracks tool within AVA.

The first message from Tracks is to initiate contact with IDA Pro, which causes IDA Pro to send back the path to the XML file describing static control flow. Next the Tracks framework can receive navigation, debugging and renaming events from IDA Pro, which contain additional information about functions. This information includes the index of the call (8 in this case), the function's address, the function's name, the name of the external file where the function resides (if applicable), and the executable file name (in this case, `calc.exe`). The executable file name is needed when there are multiple IDA Pro instances open, so that we interact with the correct one. The top requirement from the requirements elicitation in the malware group (380 out of 400) was the ability for IDA Pro to handle more than one executable at a time, so it is important that wherever possible we keep this in mind.

Additionally, we can send events to all IDA Pro instances such as enable/disable tracing messages, enable/disable tracing calls within a library module, update preference count for loops, disable tracing of a specific loop and send goodbye messages.

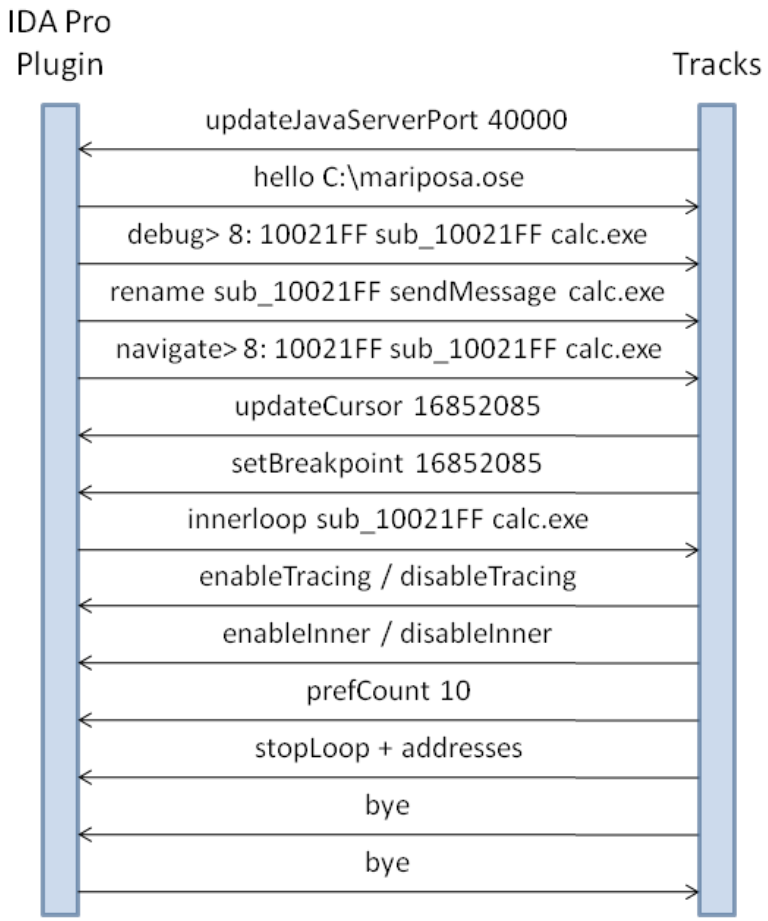

Figure 6.2: Messages between IDA Pro and Tracks.

6.2 Obtaining Data

In order to create tools, we first needed to define a model for the data we would need. The initial project proposal anticipated the solution would include an intermediate common format for multiple assembly languages. Ultimately, no existing intermediate language (IL) could support our goals due to fundamental differences between the languages [71]. Therefore, each of our tools use their own home-grown XML format specific to their needs. We discuss how data was obtained and the XML format of such data by both assembly languages: mainframe code written in HLASM, and disassembled assembly in x86.

6.2.1 Data for Mainframe (HLASM)

Due to limited access to mainframe code, one of our research partners at CA labs created tools to scrape mainframe assembler listing files to extract the necessary data. The data provision was planned in phases as shown in Table 6.1. Phase one involved the extraction of a list of modules in the system, their CSECTs and DSECTs as well as external symbols. This phase was planned to aid in building tools based on the architecture. Phase two included which modules are included by a given module, which would further complete the package structure. Phase three dealt with code shared among CSECTs. Phase four contained the data necessary to reason about control flow.

Phase	Data
One	Modules, CSECTs, DSECTs and External Symbols
Two	Modules, Macros, Copybooks and How They are Applied
Three	Combination of Macros and CSECTs
Four	Procedures, Calls and Entry Points

Table 6.1: Data Phases for CA Labs.

Unfortunately our contact at CA who was providing us with this data left the company and we were therefore not able to obtain data for all four phases. We were able to obtain some data from phase one which was fed into our LegaSee tool, as well as preliminary data from phase four for control flow for Tracks. Since access to CA programs was limited due to intellectual property constraints, we instead used data obtained from large open source projects written in HLASM that were comparable in size to CA Labs' projects. We use two such systems in this dissertation. The first is an MVS program called CBT019 [104], otherwise known as FOOD LION Utilities by John Hooper, and the second is an Algol package [105].

This data was produced through python scripts that directly parse the textual listing files. The assembler listing files are produced by compiling the system which produces files which are a mixture of the original assembler as well as what is produced by the compiler. A snippet of the listing file for CBT019 is shown in Figure 6.4, while the corresponding assembly code for the same portion of the listing is shown in Figure 6.3.

While a correct approach would be based off of the ADATA file [106] produced by the compiler, this would have required assembly language programming by our contact, who rather preferred python as an immediate solution. The scripts created

```
*********************************************************************
*                                                                   *
*         OPEN THE FILE FOR INPUT, AND GET READY TO PROCESS         *
*         EACH MEMBER                                               *
*                                                                   *
*********************************************************************
OPENCK   OPEN  CHECKDCB             OPEN THE FILE
         L     R10,TBLADDR          LOAD ADDRESS OF TABLE
FIND     FIND  CHECKDCB,0(R10),D    POSITION TO MEMBER
         MVC   MEMBER,0(R10)        SAVE MEMBER NAME
READBLK  L     R6,BUFADDR           LOAD ADDRESS OF INPUT BUFFER
         READ  DECB1,SF,CHECKDCB,(R6),'S' READ BLOCK FROM MEMBER
         CHECK DECB1                WAIT FOR I/O TO COMPLETE
         LA    R9,CHECKDCB          LOAD ADDRESS OF DCB
         USING IHADCB,R9            SET ADDRESSABILITY TO DCB
         LH    R14,DCBBLKSI         LOAD BLOCKSIZE
         L     R15,DECB1+16         LOAD IOB ADDRESS
         SH    R14,14(,R15)         SUBTRACT RESIDUAL BYTE COUNT
         SRDA  R14,32               SHIFT TO ODD REG FOR DIVIDE
         D     R14,=F'80'           DIVIDE BY LRECL FOR RECORD COUNT
         LTR   R14,R14              TEST REMAINDER
         BZ    FIND1                ZERO, OK
         WTO   'BSCN005E ACTUAL BLOCKSIZE NOT A MULTIPLE OF LRECL',    X
               ROUTCDE=11
```

Figure 6.3: HLASM Snippet for BLKSCAN Module in CBT019.

intermediate structures in memory, in files accessible through seeks and in a number of database tables. Tables and seekable files were used because the code analyzed has roughly 900,000 lines of assembly code. By several passes through structures, the script resolve code entities and built their relationships. The raw XML data for CBT019, produced from these python scripts, is shown in Figure 6.5.

From this raw XML data, we used XSL translations to create the specific XML files for both Tracks and LegaSee. Figure 6.6 shows a snippet of the data transformed for CBT019 which provides the control flow information for our Tracks tool. One of the important challenges within HLASM was there not being an explicit concept of a subroutine call. "Subroutine calls" were then based on conventions of how registers were used, which registers were used, and typical patterns of code around this. Basically, if a branch instruction saves the next address and makes a jump to another address (typically instructions like BAL, BALR, BASR), then the location is classified as a subroutine call. If an address is the target address of more such locations, then the target address is classified as a subroutine entry point. However, code classified as a subroutine should end with a branch instruction that transfers control back to the saved address (saved by the calling branch instruction). It is important to note

```
        SEARCH PARTITIONED DATA SET FOR BLOCK SIZES                                                    Page   11
  Active Usings: BLKSCAN,R11,R12
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                    HLASM R5.0  2008/04/28 09.49
                                   294 ************************************************************************
                                   295 *                                                                      *
                                   296 *        OPEN THE FILE FOR INPUT, AND GET READY TO PROCESS             *
                                   297 *        EACH MEMBER                                                   *
                                   298 *                                                                      *
                                   299 ************************************************************************
                                   300 OPENCK   OPEN  CHECKDCB             OPEN THE FILE
000230                             301+         CNOP  0,4                          ALIGN LIST TO FULLWORD      01-OPEN
000230 4510 B238            00238  302+OPENCK   BAL   1,*+8                        LOAD REG1 W/LIST ADDR. @L2A 01-OPEN
000234 80                          303+         DC    AL1(128)                     OPTION BYTE                 01-OPEN
000235 000C6C                      304+         DC    AL3(CHECKDCB)                DCB ADDRESS                 01-OPEN
000238 0A13                        305+         SVC   19                           ISSUE OPEN SVC              01-OPEN
00023A 58A0 B8C8            008C8  306          L     R10,TBLADDR            LOAD ADDRESS OF TABLE
                                   307 FIND     FIND  CHECKDCB,0(R10),D      POSITION TO MEMBER
00023E 4110 BC6C            00C6C  309+FIND     LA    1,CHECKDCB                   LOAD PARAMETER REG 1   02-IHBIN
000242 410A 0000            00000  310+         LA    0,0(R10)                     LOAD PARAMETER REG 0   02-IHBIN
000246 1211                        311+         LTR   1,1                          IS DCB ADDRESS 0       01-FIND
000248 4770 B258            00258  312+         BNZ   *+16                         NO, BRANCH             01-FIND
00024C 41F0 0008            00008  313+         LA    15,8                         SET RETURN CODE        01-FIND
000250 4100 0010            00010  314+         LA    0,16                         SET REASON CODE        01-FIND
000254 47F0 B25C            0025C  315+         B     *+8                          BR AROUND FIND SVC     01-FIND
000258 1311                        316+         LCR   1,1                          INDICATE TYPE D        01-FIND
00025A 0A12                        317+         SVC   18                           ISSUE FIND SVC         01-FIND
00025C D207 B931 A000 00931 00000  318          MVC   MEMBER,0(R10)          SAVE MEMBER NAME
000262 5860 B8CC            008CC  319 READBLK  L     R6,BUFADDR             LOAD ADDRESS OF INPUT BUFFER
                                   320          READ  DECB1,SF,CHECKDCB,(R6),'S' READ BLOCK FROM MEMBER
000266 0700                        322+         CNOP  0,4                                                  02-IHBRD
000268 4510 B280            00280  323+         BAL   1,*+24                 LOAD DECB ADDRESS             02-IHBRD
00026C 00000000                    324+DECB1    DC    F'0'                         EVENT CONTROL BLOCK    02-IHBRD
000270 80                          325+         DC    X'80'                        TYPE FIELD             02-IHBRD
000271 80                          326+         DC    X'80'                        TYPE FIELD             02-IHBRD
000272 0000                        327+         DC    AL2(0)                       LENGTH                 02-IHBRD
000274 0000006C                    328+         DC    A(CHECKDCB)                  DCB ADDRESS            02-IHBRD
000278 00000000                    329+         DC    A(0)                         AREA ADDRESS           02-IHBRD
00027C 00000000                    330+         DC    A(0)                         RECORD POINTER WORD    02-IHBRD
000280 5061 000C            0000C  331+         ST    R6,12(1,0)                   STORE AREA ADDRESS     02-IHBRD
000284 58F0 1008            00008  332+         L     15,8(,1)                LOAD DCB ADDR          @01M 02-IHBRD
000288 BFF7 F031            00031  333+         ICM   15,B'0111',49(15)       LOAD RDWR ROUTINE ADDR @01M 02-IHBRD
00028C 05EF                        334+         BALR  14,15                   LINK TO RDWR ROUTINE   @L1C 02-IHBRD
                                   335          CHECK DECB1                  WAIT FOR I/O TO COMPLETE
00028E 4110 B26C            0026C  337+         LA    1,DECB1                      LOAD PARAMETER REG 1   02-IHBIN
000292 58E0 1008            00008  338+         L     14,8(0,1)               PICK UP DCB ADDR               01-CHECK
000296 1BFF                        339+         SR    15,15                                        @01A 01-CHECK
000298 BFF7 E035            00035  340+         ICM   15,B'0111',53(14)       LOAD CHECK ROUTINE ADDR @01C 01-CHECK
00029C 05EF                        341+         BALR  14,15                   LINK TO CHECK ROUTINE          01-CHECK
00029E 4190 BC6C            00C6C  342          LA    R9,CHECKDCB            LOAD ADDRESS OF DCB
                 R:9  00000        343          USING IHADCB,R9              SET ADDRESSABILITY TO DCB
0002A2 48E0 903E            0003E  344          LH    R14,DCBBLKSI           LOAD BLOCKSIZE
0002A6 58F0 B27C            0027C  345          L     R15,DECB1+16           LOAD IOB ADDRESS
0002AA 4BE0 F00E            0000E  346          SH    R14,14(,R15)           SUBTRACT RESIDUAL BYTE COUNT
0002AE 8EE0 0020            00020  347          SRDA  R14,32                 SHIFT TO ODD REG FOR DIVIDE
0002B2 5DE0 B7A8            007A8  348          D     R14,=F'80'             DIVIDE BY LRECL FOR RECORD COUNT
0002B6 12EE                        349          LTR   R14,R14                TEST REMAINDER
0002B8 4780 B30C            0030C  350          BZ    FIND1                  ZERO, OK
                                   351          WTO   'BSCN005E ACTUAL BLOCKSIZE NOT A MULTIPLE OF LRECL',    X
```

Figure 6.4: Listing Snippet for BLKSCAN Module in CBT019.

```
<softwarePackage name=''CBT019''>
  <copybook name=''$HASPGBL''>
  </copybook>
  <copybook name=''SMFDSECT''>
  </copybook>
  <sourceModule name=''BLKSCAN'' language=''ASM'' lastAddress=''000D20''
      isIgnored=''false'' description=''''>
    <controlSection name=''BLKSCAN'' startAddress=''00000'' length=''00D24''
        stmt=''80''>
      <addressPoints>
        <addressPoint id=''BLKSCAN@BLKSCAN.000000'' label=''BLKSCAN''
            section=''BLKSCAN'' hexOffset=''000000''>
          <callToAddress id=''DETAIL@BLKSCAN.000448''></callToAddress>
        </addressPoint>
        <addressPoint id=''None@BLKSCAN.0002A9'' label=''None'' section=''BLKSCAN''
            hexOffset=''0002A9''>
        </addressPoint>
        <addressPoint id=''HEADINGS@BLKSCAN.0003F4'' label=''HEADINGS''
            section=''BLKSCAN'' hexOffset=''0003F4''>
          <callToAddress id=''None@BLKSCAN.000CF5''></callToAddress>
        </addressPoint>
        <addressPoint id=''DETAIL@BLKSCAN.000448'' label=''DETAIL''
            section=''BLKSCAN'' hexOffset=''000448''>
          <callToAddress id=''None@BLKSCAN.000CF5''></callToAddress>
        </addressPoint>
        <addressPoint id=''SCAN@BLKSCAN.00049E'' label=''SCAN'' section=''BLKSCAN''
            hexOffset=''00049E''>
          <callToAddress id=''DETAIL@BLKSCAN.000448''></callToAddress>
        </addressPoint>
        <addressPoint id=''None@BLKSCAN.000C3D'' label=''None'' section=''BLKSCAN''
            hexOffset=''000C3D''>
        </addressPoint>
        <addressPoint id=''None@BLKSCAN.000CA5'' label=''None'' section=''BLKSCAN''
            hexOffset=''000CA5''>
        </addressPoint>
        <addressPoint id=''None@BLKSCAN.000CF5'' label=''None'' section=''BLKSCAN''
            hexOffset=''000CF5''>
        </addressPoint>
      </addressPoints>
    </controlSection>
    <dummySection name=''IHADCB'' startAddress=''00000'' length=''00058''>
    </dummySection>
  </sourceModule>
```

Figure 6.5: CSECT and DSECT Data for CBT019.

that some uses of BAL, BALR and BASR serve completely different purposes than subroutine calls. We see that stating whether or not something is a subroutine is non-trivial and will not be 100% accurate. Since our contact at CA left at this stage of data extraction, our current control flow information follows a set of over-simplified classification rules for subroutines. While this may lead to some confusing situations with some data, it is an issue that can be fixed given future work on listing parsing, without any changes to the control flow tool itself.

Figure 6.7 shows a snippet of the XML produced for the LegaSee tool through the XSLT. This data pertains to CSECTs and DSECTs only.

6.2.2 Data for Malware (x86)

IDA Pro provides an extensive API that allows users to extend it through plugins, or by running python scripts directly from the plugin menu. Since IDA Pro was the primary tool used by all malware analysts, we leverage this API in order to extract the necessary data. The original DRDC project that looked at control flow within IDA Pro was called Opening up Architectures of Software-Intensive Systems (OASIS). In particular, the OASIS Sequence Explorer (OSE) [107] contained an IDA Pro plugin that gathered control flow information, and dumped this information to a text file. Figure 6.8 shows the output file for the Eclipse executable.

The plugin works by going through the binary and first listing all entry points into that binary. There is usually only one for an executable, but DLLs can have several. Then for each function in the binary, the plugin lists every call made. It is only one level deep, as opposed to an execution trace. The first line is an entry point, denoted by ">>", which contains the function address and function name of that entry point. Sections delimited by "<" and ">" indicate the start of calls for a particular function, and the information is the same for that of an entry point (address, name). Within this section are all the calls made within that function which also contain the function address and name. If the call line starts with a "-1", then it indicates a call to an external function and the binary which contains this function is listed at the end of the call line. A call line that begins with BADADDR indicates that retrieving that information failed, usually meaning that the call is to an address contained in a register. Otherwise the first number indicates the index.

We extended this functionality through our own plugin that similarly retrieves control flow information from IDA Pro. However we use XML as our format, and

```
<softwarePackage name=''CBT019''>
  <copybook name=''$HASPGBL''>
  </copybook>
  <copybook name=''SMFDSECT''>
  </copybook>
  <section name=''BLKSCAN''>
      <functionEntryPoint address=''00000'' section=''BLKSCAN'' name=''BLKSCAN''
          index=''00D24''>
         <function address=''000000'' name=''BLKSCAN'' index=''1''
             section=''BLKSCAN''>
            <call calladdress=''000448'' name=''DETAIL'' functionaddress=''000448''
                index=''local''
                  externalFile=''BLKSCAN''/>
         </function>
         <function address=''0002A9'' name=''None'' index=''2'' section=''BLKSCAN''/>
         <function address=''0003F4'' name=''HEADINGS'' index=''3''
             section=''BLKSCAN''>
            <call calladdress=''000CF5'' name=''None'' functionaddress=''000CF5''
                index=''local''
                  externalFile=''BLKSCAN''/>
         </function>
         <function address=''000448'' name=''DETAIL'' index=''4''
             section=''BLKSCAN''>
            <call calladdress=''000CF5'' name=''None'' functionaddress=''000CF5''
                index=''local''
                  externalFile=''BLKSCAN''/>
         </function>
         <function address=''00049E'' name=''SCAN'' index=''5'' section=''BLKSCAN''>
            <call calladdress=''000448'' name=''DETAIL'' functionaddress=''000448''
                index=''local''
                  externalFile=''BLKSCAN''/>
         </function>
         <function address=''000C3D'' name=''None'' index=''6'' section=''BLKSCAN''/>
         <function address=''000CA5'' name=''None'' index=''7'' section=''BLKSCAN''/>
         <function address=''000CF5'' name=''None'' index=''8'' section=''BLKSCAN''/>
      </functionEntryPoint>
  </section>
  <section name=''CA''>
      <functionEntryPoint address=''0'' section=''TFVSENV'' name=''TFVSENV''
          index=''0''/>
      <functionEntryPoint address=''0'' section=''OPSAMD'' name=''OPSAMD''
          index=''0''/>
      <functionEntryPoint address=''0'' section=''TFVSEOPE'' name=''TFVSEOPE''
          index=''0''/>
      <functionEntryPoint address=''0'' section=''TFVSESR'' name=''TFVSESR''
          index=''0''/>
      <functionEntryPoint address=''0'' section=''TSIDSYS'' name=''TSIDSYS''
          index=''0''/>
      <functionEntryPoint address=''0'' section=''TFVSERR'' name=''TFVSERR''
          index=''0''/>
   </section>
  <section name=''CHECKPVT''>
      <functionEntryPoint address=''00000'' section=''CHECKPVT'' name=''CHECKPVT''
          index=''001D2''/>
   </section>
```

Figure 6.6: Static Control Flow Information for CBT019.

```
<softwarePackage name=''CBT019''>
  <sourceModule name=''BLKSCAN'' language=''ASM'' lastAddress=''000D20''
      isIgnored=''false'' description=''''>
    <controlSection name=''BLKSCAN'' startAddress=''00000'' length=''00D24''
        stmt=''80''/>
    <dummySection name=''IHADCB'' startAddress=''00000'' length=''00058''/>
  </sourceModule>
  <sourceModule name=''CHECKPVT'' language=''ASM'' lastAddress=''0001C8''
      isIgnored=''false'' description=''''>
    <controlSection name=''CHECKPVT'' startAddress=''00000'' length=''001D2''
        stmt=''65''/>
    <dummySection name=''CVT'' startAddress=''00000'' length=''00500''/>
    <dummySection name=''CVTXTNT1'' startAddress=''00000'' length=''0000C''/>
    <dummySection name=''CVTVSTGX'' startAddress=''00000'' length=''00050''/>
    <dummySection name=''CVTXTNT2'' startAddress=''00000'' length=''00084''/>
  </sourceModule>
</softwarePackage>
```

Figure 6.7: LegaSee XML Format.

we also additionally provide the address at which the call to a function is made. Figure 6.9 shows the XML output created for the calculator executable included with Windows.

While this approach satisfied static control flow data requirements, it is not a solution for dynamic traces, or user navigation. Therefore we created another XML format specific to these dynamic control graphs. This format is similar to the above but maintains the order of calls, as well as multiple calls to the same function which are not available for static graphs. This format is shown in Figure 6.10.

6.3 Tracks: Sequence Diagrams for Assembly

One of the most difficult challenges identified in our surveys and requirements elicitation, is to follow control flow within assembly due to the inherently unstructured nature of assembly code (see Sections 5.1.6 and 5.2).

Tracks is built on top of the Diver framework which was introduced in Section 2.3.1. To summarize, Diver is an open-source and extensible sequence diagram tool built using the Eclipse framework [73] by the CHISEL Group at the University of Victoria. Diver provides features for extremely large traces, for example, users may set any of the functions as the root of the diagram to reduce the amount of information displayed. There are breadcrumbs at the top of the diagram to navigate back to the previous view. There is also a thumbnail outline pane to quickly navigate

```
>> 57:   404907   start
> 0:     401000   _wmain
         -1:      405020   ds:GetVersionExW         KERNEL32.dll
         -1:      405068   ds:malloc        MSVCRT.dll
         -1:      405040   ds:WideCharToMultiByte   KERNEL32.dll
         -1:      405068   ds:malloc        MSVCRT.dll
         -1:      405040   ds:WideCharToMultiByte   KERNEL32.dll
         10:      401BC0   sub_401BC0
         -1:      405064   ds:free MSVCRT.dll
         -1:      405064   ds:free MSVCRT.dll
         1:       401188   sub_401188
< 0
> 1:     401188   sub_401188
         -1:      40508C   ds:setlocale     MSVCRT.dll
         -1:      405068   ds:malloc        MSVCRT.dll
         58:      404A18   memcpy
         -1:      405084   ds:wcschr        MSVCRT.dll
         2:       401539   sub_401539
         4:       40173D   sub_40173D
         20:      402691   sub_402691
         19:      4025D0   sub_4025D0
         3:       401629   sub_401629
         3:       401629   sub_401629
         5:       401796   sub_401796
         -1:      405080   ds:_wcsdup       MSVCRT.dll
         8:       4018C9   sub_4018C9
         6:       401831   sub_401831
         9:       401949   sub_401949
         34:      4037A3   sub_4037A3
         -1:      40507C   ds:wcslen        MSVCRT.dll
         -1:      40507C   ds:wcslen        MSVCRT.dll
         -1:      405068   ds:malloc        MSVCRT.dll
         -1:      405078   ds:swprintf      MSVCRT.dll
         32:      4036D3   sub_4036D3
         -1:      405070   ds:fwprintf      MSVCRT.dll
         -1:      405064   ds:free MSVCRT.dll
         -1:      40506C   ds:exit MSVCRT.dll
         36:      4037C1   sub_4037C1
         BADADDR [ebp+var_2C]
         32:      4036D3   sub_4036D3
         -1:      405070   ds:fwprintf      MSVCRT.dll
         -1:      40506C   ds:exit MSVCRT.dll
         36:      4037C1   sub_4037C1
         BADADDR [ebp+var_10]
         32:      4036D3   sub_4036D3
         -1:      405070   ds:fwprintf      MSVCRT.dll
         -1:      40506C   ds:exit MSVCRT.dll
         35:      4037B2   sub_4037B2
         -1:      405064   ds:free MSVCRT.dll
         -1:      405064   ds:free MSVCRT.dll
         -1:      405064   ds:free MSVCRT.dll
         -1:      405064   ds:free MSVCRT.dll
< 1
```

Figure 6.8: OASIS Sequence Explorer Output for Eclipse.exe

```
<sourcecode filename=''calc.exe.ose''>
   <functionEntryPoint address=''1009768'' index=''250'' module=''calc.exe'' name=''start''>
      <function address=''1001635'' index=''0'' module=''calc.exe'' name=''sub_1001635''>
         <call calladdress=''10016A0'' externalfile=''KERNEL32.dll'' functionaddress=''1001194''
               index=''external''
               module=''calc.exe''
               name=''GetModuleHandleW''/>
         <call calladdress=''10016A3'' externalfile=''USER32.dll'' functionaddress=''10013F4''
               index=''external''
               module=''calc.exe''
               name=''LoadStringW''/>
         <call calladdress=''10016B7'' externalfile=''calc.exe'' functionaddress=''100943C''
               index=''244''
               module=''calc.exe''
               name=''sub_100943C''/>
         <call calladdress=''10016D0'' externalfile=''calc.exe'' functionaddress=''1009337''
               index=''238''
               module=''calc.exe''
               name=''sub_1009337''/>
         <call calladdress=''10016E2'' externalfile=''KERNEL32.dll'' functionaddress=''1001194''
               index=''external''
               module=''calc.exe''
               name=''GetModuleHandleW''/>
         <call calladdress=''10016E5'' externalfile=''calc.exe'' functionaddress=''1009E97''
               index=''260''
               module=''calc.exe''
               name=''sub_1009E97''/>
         <call calladdress=''100171D'' externalfile=''calc.exe'' functionaddress=''1009405''
               index=''243''
               module=''calc.exe''
               name=''sub_1009405''/>
         <call calladdress=''100172A'' externalfile=''calc.exe'' functionaddress=''1009405''
               index=''243''
               module=''calc.exe''
               name=''sub_1009405''/>
         <call calladdress=''100173B'' externalfile=''USER32.dll'' functionaddress=''10013D0''
               index=''external''
               module=''calc.exe''
               name=''LoadCursorW''/>
         <call calladdress=''1001756'' externalfile=''KERNEL32.dll'' functionaddress=''1001194''
               index=''external''
               module=''calc.exe''
               name=''GetModuleHandleW''/>
```

Figure 6.9: Static Control Flow Data for calc.exe

```
<dynamicTrace filename=''calc.exe''>
   <functionEntryPoint address='''' index=''0'' module='''' name=''''>
      <function address='''' externalfile=''User'' index=''1'' module='''' name=''User''
        stereotype=''''>
         <call act=''0'' calladdress=''100739D'' externalfile='''' functionaddress=''100739D''
               index=''79''
               module=''calc.exe''
               name=''_WinMainCRTStartup''/>
      </function>
      <function act=''0'' address=''100739D'' externalfile='''' index=''2'' module=''calc.exe''
                name=''_WinMainCRTStartup''
                stereotype=''''>
         <call act=''0'' calladdress=''1007568'' externalfile='''' functionaddress=''1007568''
               index=''80''
               module=''calc.exe''
               name=''__SEH_prolog''/>
         <call act=''0'' calladdress=''7C80B741'' externalfile=''KERNEL32.DLL''
               functionaddress=''7C80B741''
               index=''-1''
               module=''calc.exe''
               name=''GetModuleHandleA''/>
         <call act=''0'' calladdress=''77C3537C'' externalfile=''MSVCRT.DLL''
               functionaddress=''77C3537C''
               index=''-1''
               module=''calc.exe''
               name=''__set_app_type''/>
         <call act=''0'' calladdress=''77C1F1DB'' externalfile=''MSVCRT.DLL''
               functionaddress=''77C1F1DB''
               index=''-1''
               module=''calc.exe''
               name=''__p__fmode''/>
         <call act=''0'' calladdress=''77C1F1A4'' externalfile=''MSVCRT.DLL''
               functionaddress=''77C1F1A4''
               index=''-1''
               module=''calc.exe''
               name=''__p__commode''/>
         <call act=''0'' calladdress=''10075F4'' externalfile='''' functionaddress=''10075F4''
               index=''86''
               module=''calc.exe''
               name=''__setargv''/>
         <call act=''0'' calladdress=''10075DD'' externalfile='''' functionaddress=''10075DD''
               index=''85''
               module=''calc.exe''
               name=''__setdefaultprecision''/>
         <call act=''0'' calladdress=''10075D2'' externalfile='''' functionaddress=''10075D2''
               index=''84''
               module=''calc.exe''
               name=''__initterm''/>
         <call act=''0'' calladdress=''77C1EEEB'' externalfile=''MSVCRT.DLL''
               functionaddress=''77C1EEEB''
               index=''-1''
               module=''calc.exe''
               name=''__getmainargs''/>
      </function>
```

Figure 6.10: Dynamic Control Flow Data for calc.exe

around the diagram.

We extended Diver by defining our own content and label providers for each of the sequence diagrams (both static and dynamic), as well as implementing additional features such as a function tree view, saving diagram state, detecting cycles within the diagram, showing external calls and custom functionality for events (such as navigation and setting breakpoints). Table 6.2 outlines the feature additions that are made by our Tracks tool. Ellipses indicate where further extensions may have been provided by Diver, in addition to the examples for Java which are included in the Diver project.

Tracks was initially developed to show both static and dynamic sequence diagrams, as well as navigation histories. However, it was later extended to provide MSDN documentation, and collaboration and documentation support. Tracks was the winner of IDA Pro's Plugin Contest in 2011 [108]. A demo video of Tracks is available online at: `www.jenniferbaldwin.info/diva/presentations.html`.

This section presents the features of Tracks including the three distinct views: static traces, dynamic traces and navigation histories. We additionally discuss the MSDN documentation tie-in, and the collaboration/documentation features. Finally, we discuss how Tracks supports multiple IDA Pro instances. This was necessary because IDA Pro is single-threaded and we may need to debug more than one executable at a time (e.g. a program and its libraries). We will also see later in the following chapter, Section 7.3.3, that the Mariposa botnet injects code into `explorer.exe` which then needs to be debugged separately in another IDA Pro instance. This means that Tracks can show a complete dynamic control flow graph, but it also means that if you double click on an element, Tracks will navigate to the correct IDA Pro instance.

6.3.1 Static View

To see the static control flow diagram, the user can select any function defined within the executable being disassembled and then view and select any call this function makes, expanding the diagram as calls are selected. Additionally the user can right-click on any activation and choose to expand all calls within the diagram. This static view shows every call that could possibly be made from a function. However it does not guarantee ordering, and does not show calls to the same function that are made more than once. The XML data format used for static views within Tracks was discussed previously in Section 6.2, and shown specifically for the calculator

Feature	Diver	Tracks
Providers	Java ...	Static Assembly Dynamically built Assembly Trace Reversed
Save Diagram		Root, expanded calls and current location Save diagram to XML (dynamic to trace)
System API Calls		Icon indication and use of module names
Loops and Cycles	Cycle detection	Adapted cycle detection Loop detection (based on preference)
Collaboration and Documentation		Comment threads Stars added to indicate comments Color coding of cycles based on number of comments
Diagram Editing		Calls can be pruned
Integration with IDA Pro		Included: - data, navigation, breakpoints - multiple exe diagrams supported
Hierarchy View	Java Package Explorer ...	Assembly View based on model

Table 6.2: Diver versus Tracks: Feature Comparison.

executable in Figure 6.9.

In order for the user to select a function to view the static diagram for, Tracks provides a tree view that lists all of the functions defined within an executable. This tree view is shown in Figure 6.11. By double clicking a function (or right clicking for other options), the static control flow graph for that function is opened within the Tracks sequence diagram editor. An example diagram is shown in Figure 6.12. This screenshot shows that the user has selected the function `sub_1001635`, and can then expand the function calls they are interested in to see what calls that function makes. Functions that have an `I` icon next to them are imported functions, meaning they are located in another module. At the top of the figure, there is a diagram that shows which module this function is defined in. Here we can see that many of the imported functions come from the `KERNEL32.dll` file. When an imported function is selected, the XML file corresponding to it, if it exists, is parsed and the information added to the diagram. We can also see the thumbnail view in the outline pane at the bottom. The viewer allows users to set any of the calls as the new root of the diagram and reset the root to the caller of that function. These are available as right-click options on the subroutine's lifeline. Additionally, breadcrumbs at the top of the diagram allow the user to select any function along the path to navigate through the calls.

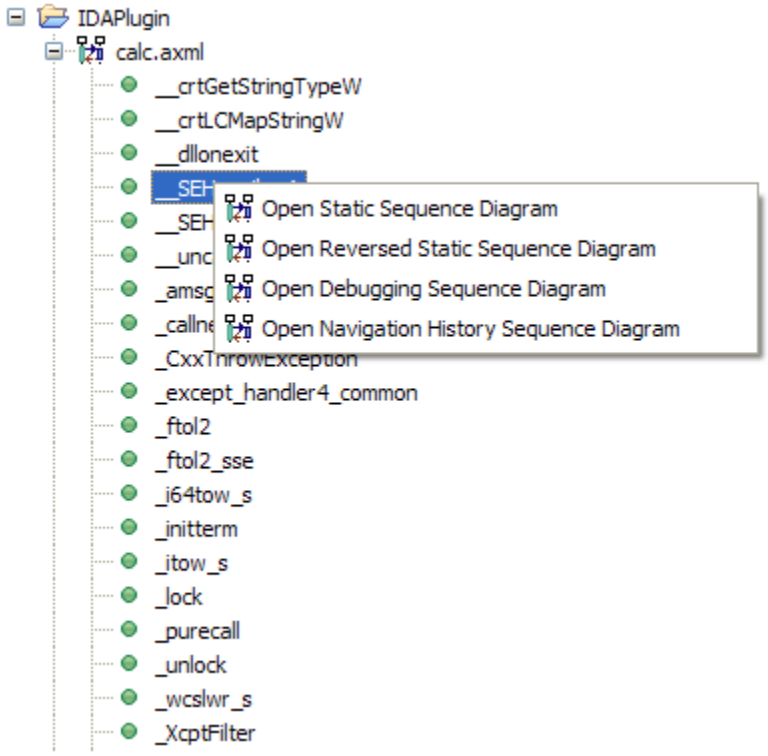

Figure 6.11: Tree View of Functions in calc.exe

If the user double clicks on an activation, IDA Pro displays that function. If the user double clicks on a call, IDA Pro shows where the call is made. It is also possible to automatically synchronize the navigation as we step through the diagram.

Finally, a user can open a reversed control flow view for a function through a right click option on the function tree. This diagram will show all possible paths that finish by calling the selected function. The interaction is similar to that of forward control flow in that users can follow the path that they are interested in. The diagram also draws from left to right, however we see that the arrow heads have been reversed to indicate that the control flow is indeed in the opposite direction. An example of this is shown in Figure 6.13.

When the user is finished with their analysis, Tracks will ask if they would like to save the state of the diagram. Should they choose to do so, Tracks will save which calls are expanded/collapsed, the current root of the diagram, as well as the visible portion of the diagram if the user has scrolled. An example of how the state is saved to a .dat file is shown in Figure 6.14.

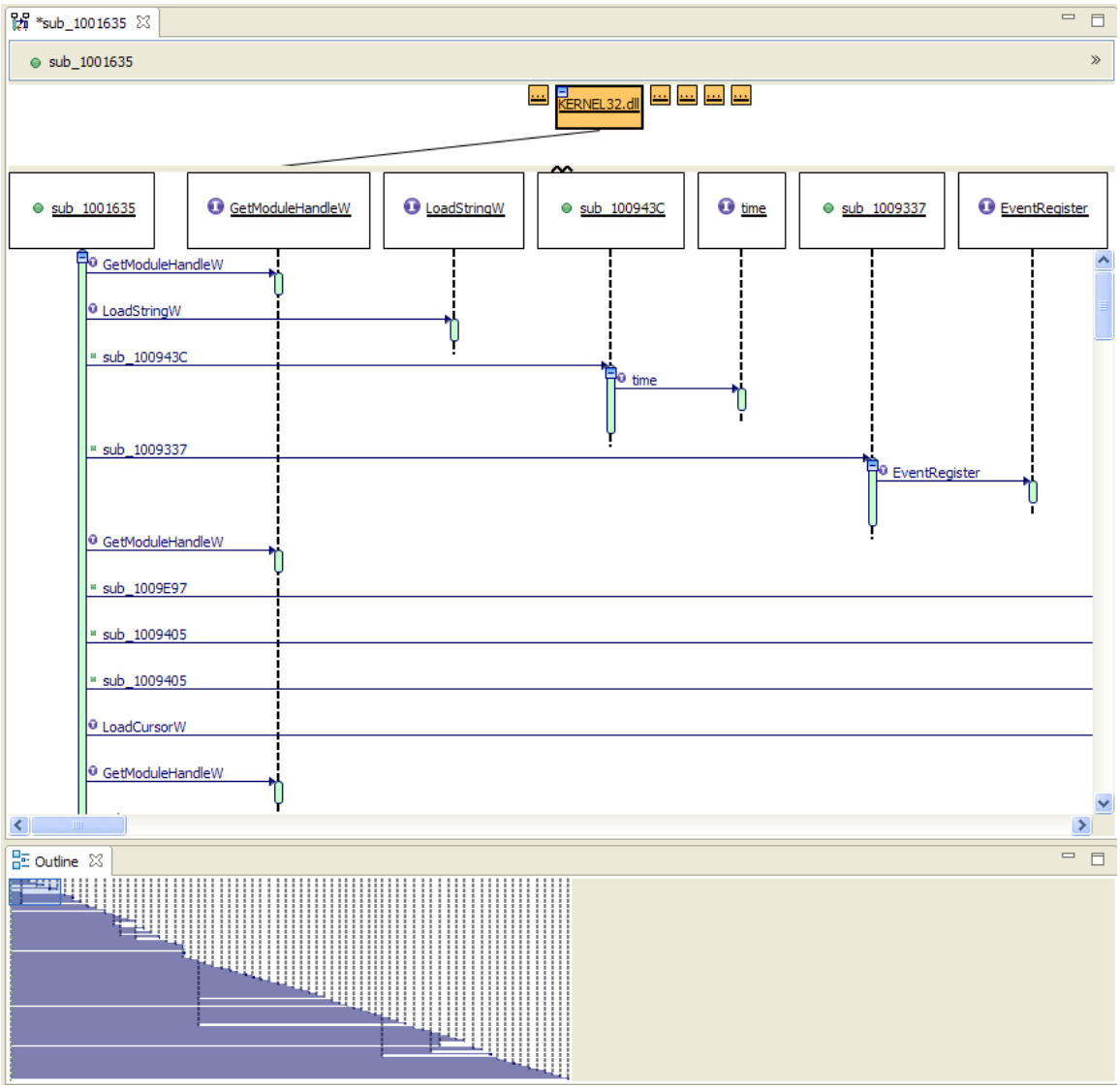

Figure 6.12: Forward Control Flow View for sub_1001635 in calc.exe

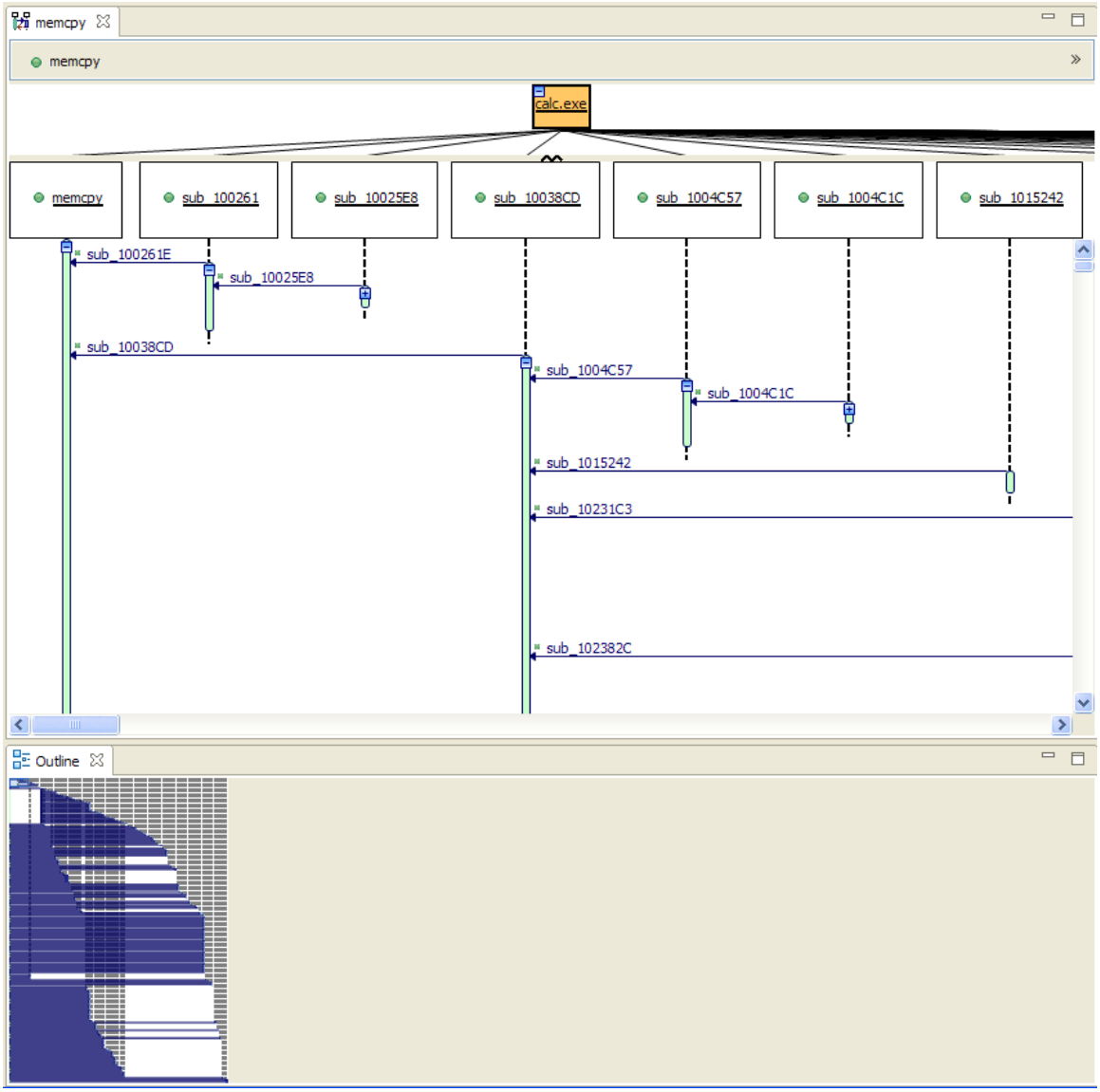

Figure 6.13: Reversed Control Flow View for memcpy Wrapper in calc.exe

```
<sequence>
   <function name=''sub_1001635''>
      <root callindex=''0'' externalfile=''calc.exe'' name=''sub_1001635''/>
      <expanded externalfile=''calc.exe'' module=''calc.exe'' name=''sub_1001635''/>
      <selection callindex=''0'' callingnode=''sub_1001635''
          externalfile=''calc.exe''
                 module=''calc.exe''
                 name=''sub_1001E8F''/>
   </function>
</sequence>
```

Figure 6.14: Diagram State Information Format.

6.3.2 Dynamic Views

Dynamic sequence diagrams are created while an executable is being debugged within IDA Pro. We have previously discussed in Section 6.1.2 how these events are generated and how the messages are received by Tracks. These messages are received whenever a new function is executed during a debugging session, and each call is then added to the correct executable's dynamic diagram. In contrast to the static view, ordering is accurate and if the same function is called twice, that is also shown. The dynamic diagram can be opened through the File menu, or as a right-click option in the function tree view. If the diagram is opened from the tree view, a breakpoint is set at that function within the corresponding IDA Pro instance. An example of a dynamic diagram is shown in Figure 6.15.

Users also have the option of setting which particular debugging events they would like to trace. These are available through Tracks' preferences which are shown in Figure 6.16. When these preferences are changed in Tracks, a message is sent to all open instances of IDA Pro. The user has two choices for this diagram: to diagram all of the calls (enables step tracing within IDA Pro), or to diagram just the calls that are stepped into. When diagramming only the calls that are stepped into, hitting a breakpoint adds that function call to the *User* lifeline. There is also an option to trace calls within imported modules.

Additionally, a loop count number can be set so that if a cycle (or loop) is encountered this number of times, it is collapsed (or colored as a loop). In this context, loops refer to iterations within a single function and cycles refer to iterations of a pattern of function calls. Loops are detected within the IDA Pro plugin and cycles are detected within Tracks. This functionality occurs in the IDA Pro plugin by recording

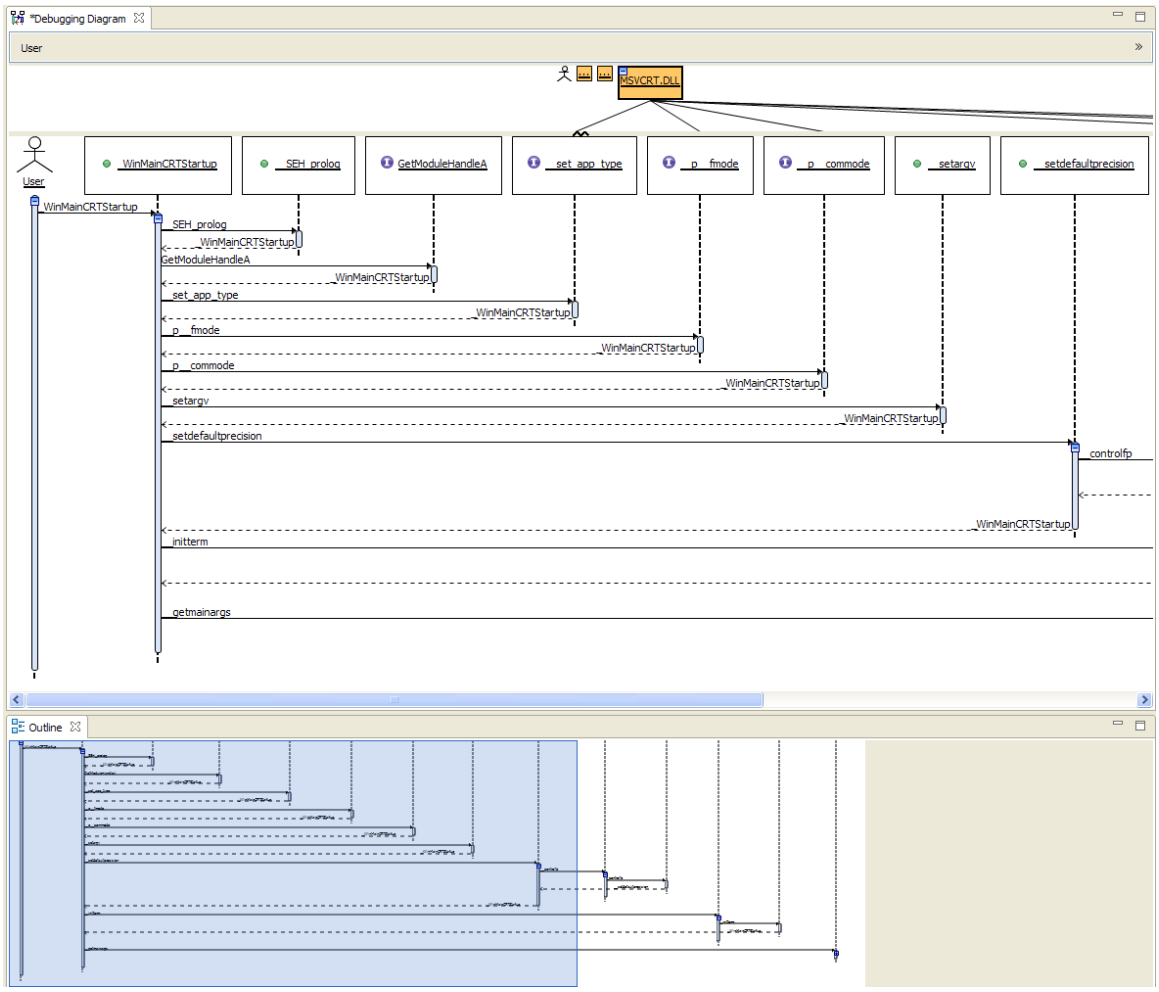

Figure 6.15: Tracks' Dynamic View.

the address whenever a command that jumps is executed. If the jumps occur within the same function and to the same address n times, where n is the preference for loop count, then a message is sent to Tracks to color the activation red.

Cycles are detected by Tracks using a simple algorithm to detect graph cycles. This algorithm works on an array of function calls, represented as strings, to find repeating strings. If a cycle is detected, it is immediately collapsed. A message is then sent to IDA Pro to stop sending messages for the cycle along with the address pattern to ignore. Examples including screenshots of loops and cycles are shown in the following chapter.

When the user is finished with the debugging session, they can choose to save the dynamic sequence diagram as a trace file so that it can be opened and analyzed at a later time. The dynamically built diagram is then exported to a list of calls in XML

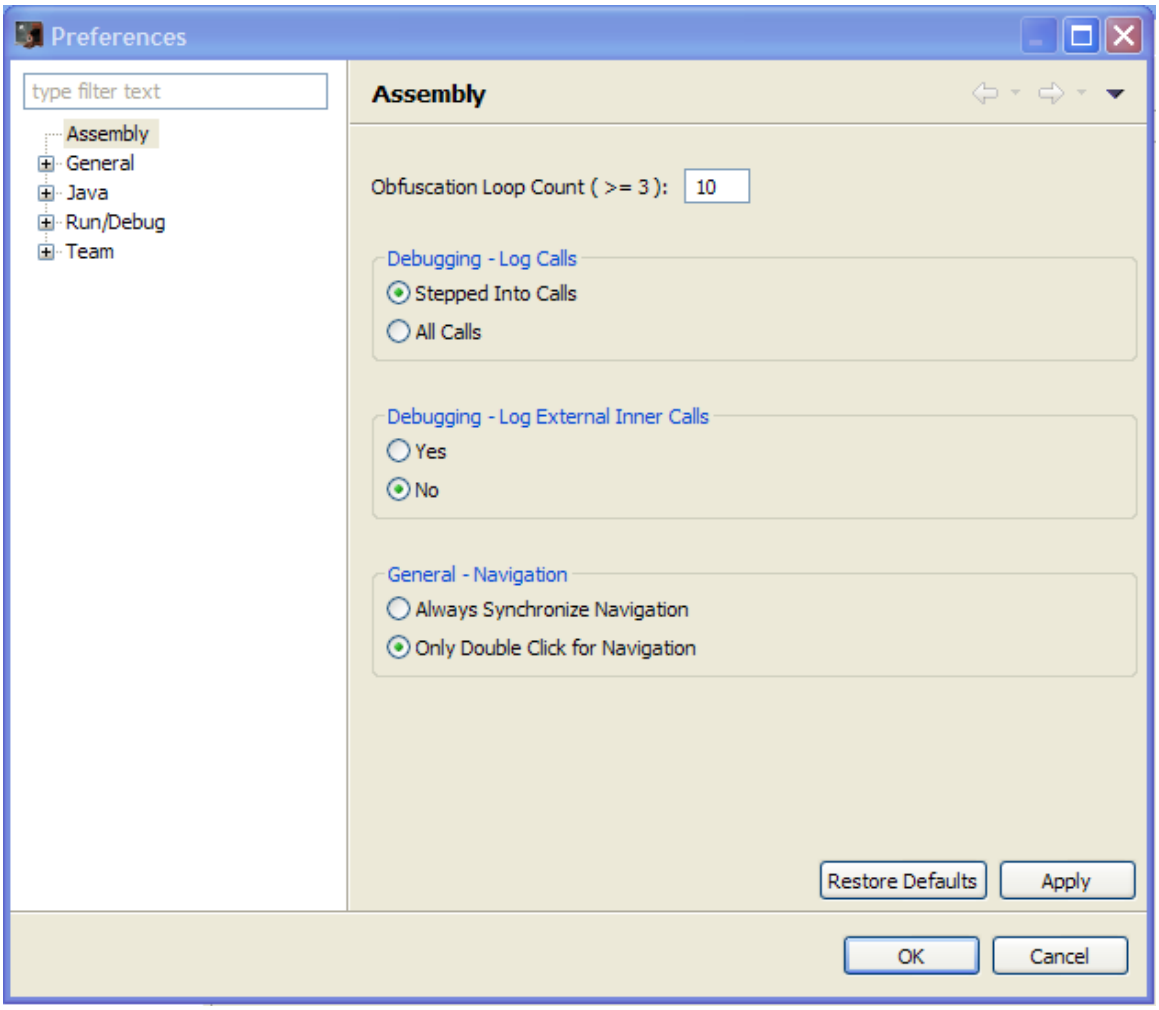

Figure 6.16: Tracks' Preferences.

format, as shown in Figure 6.10. The user can then reopen this trace file similar to the static view, as well as save the state of the diagram.

6.3.3 Navigation History View

This navigation view provides a history of the analysis a user has done within IDA Pro. This diagram is similar to the dynamic control flow diagram in that it is built dynamically and uses the *User* lifeline as the root, but it does so based on IDA Pro UI events instead of debugging events. The diagram is opened again through the File menu or by right clicking a function from the function tree. If it is selected through the function tree, then IDA Pro will automatically navigate to this function to begin analysis. Function calls are then added to the diagram as the user navigates through

the codebase within IDA Pro. Selecting a new function from the function view in IDA Pro adds it as a call from the *User* lifeline. Then, selecting a cross reference (or call) from within that function adds it as a call from that function's lifeline. An example of a navigation view is shown in Figure 6.17. This requirement came up in elicitation, see Section 4.3.4, requirement 4.

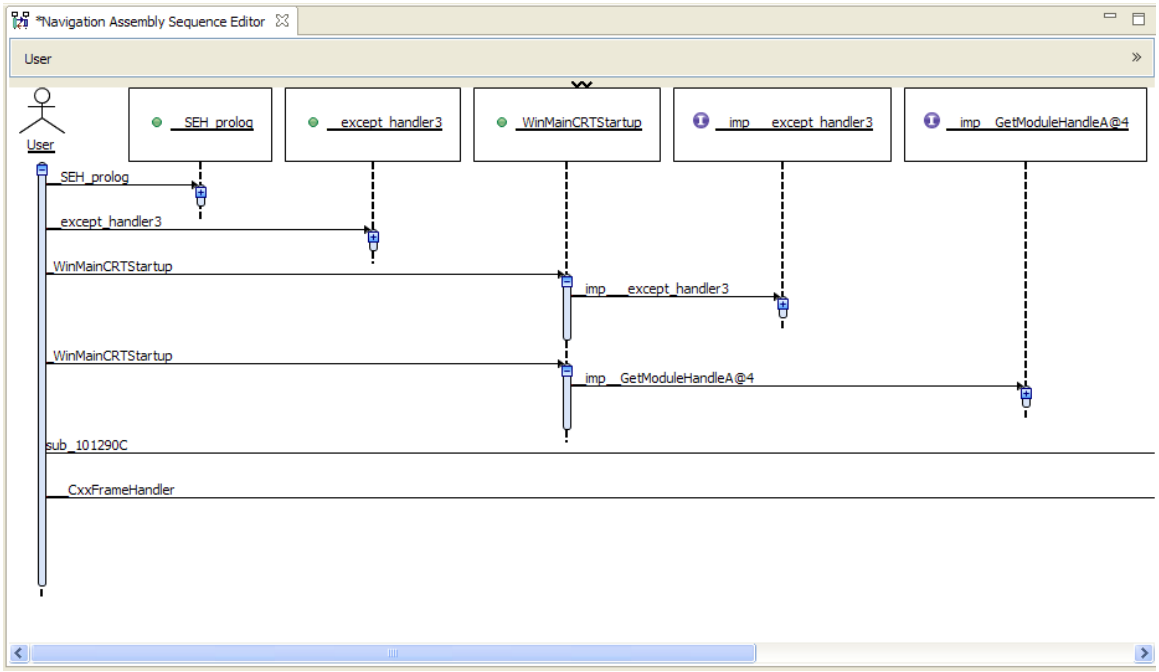

Figure 6.17: Tracks' Navigation History View.

6.3.4 MSDN Documentation

This feature was added to Tracks in direct response to a request made by the malware group both directly as well as in the requirements elicitation session (Section 4.3.4, requirement 6). The requirement was to easily access API documentation from the tool support (by visiting a web page or local help). Currently, analysts have to open a web browser and type in the function name they are looking for to retrieve the MSDN documentation. They most likely do so on a separate machine since the analysis machine cannot be connected to the Internet due to risk of spreading infection. Another option is to use the help functionality that comes installed locally in Visual Studio.

For our prototype shown in Figure 6.18, we simply embed a web browser window in a view alongside Tracks. Whenever an external function is selected and the MSDN

documentation chosen through the right-click option, we simply use Google's "I'm Feeling Lucky" search on the `msdn.microsoft.com` website to load the appropriate page in the embedded browser. This has worked correctly for all functions tried so far. The only limitation at this point is the necessity for internet access. Future plans are to download the MSDN library and index it for local use. Another potential option is to access the installed Visual Studio help.

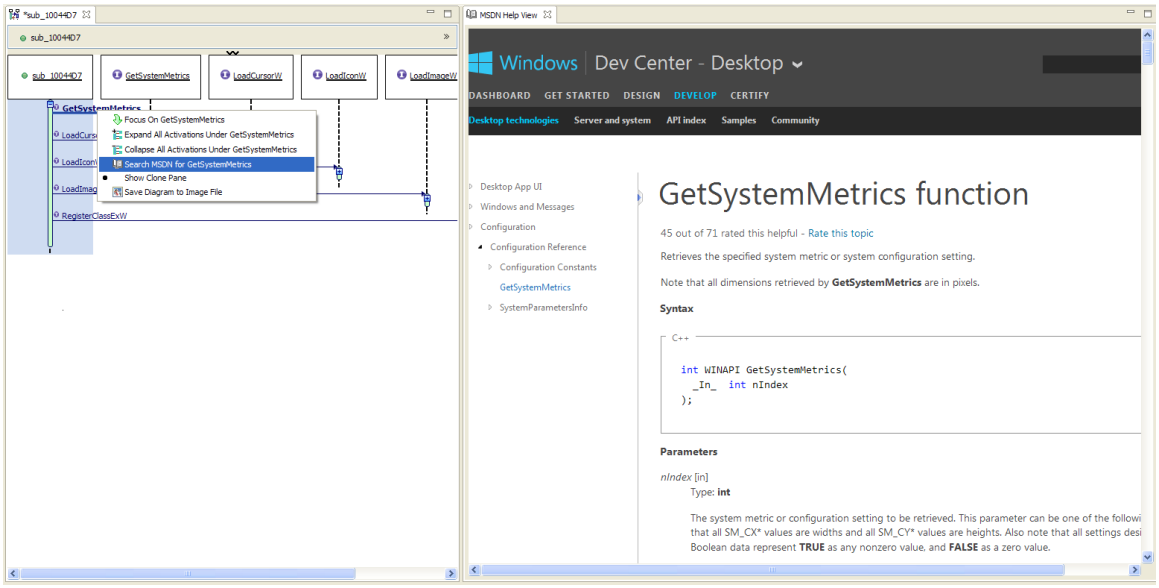

Figure 6.18: MSDN Help View in Tracks.

6.3.5 Comment Threads within Tracks

In order to address some of the issues discovered in the survey (Section 3.3), namely the lack of commenting support, including importing existing documentation and documentation of execution paths, we built a comments view into Tracks, the idea being to add comments over shared artifacts [109]. One example of this is Google's Sidewiki [110] which is a browser sidebar that lets you contribute and read information alongside any web page, shown in Figure 6.19. Users can add information such as background, tips and perspectives to annotate web content.

In order to add collaborative documentation support to Tracks, we added the ability to create comments on calls, lifelines and cycles within the sequence diagram. A preference is given to the user to import the comment data when the sequence diagram files are loaded. If comments exist, the artifacts are marked with color-

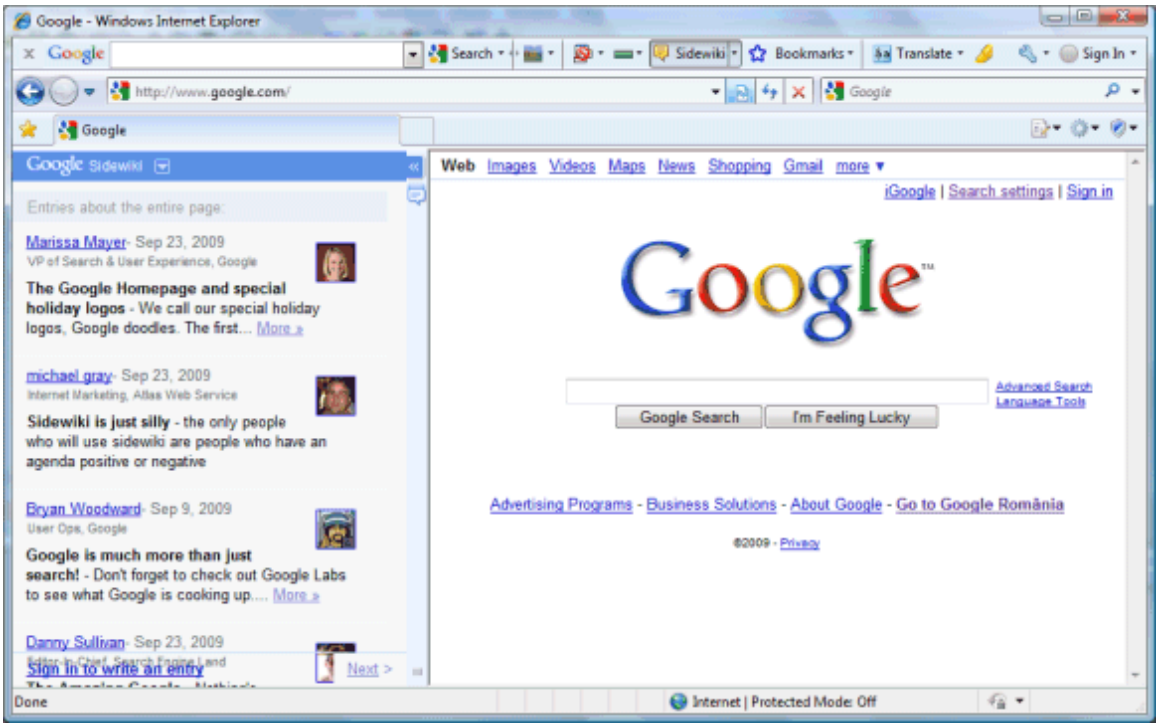

Figure 6.19: Google Sidewiki.

coded stars. Red stars indicate that there is little activity and is used for items with less than 10 comments. Yellow stars indicate a medium level of activity and are used when there are 10 or more comments but less than 25. Green stars are used to indicate high activity and are used when there are 25 or more comments. This color selection is based on the download health colors commonly used in bit torrent applications, where green represents that the file is available for download from many others.

A sequence diagram showing the comment annotations is shown in Figure 6.20. In this screenshot, we are investigating the Mariposa botnet [6, 111] and the trace file for server communication. The left panel shows the sequence diagram with comments on the two lifelines to the far right, indicated by the red stars within the lifeline boxes. The figure also displays a cycle, which is outlined in red and filled in with grey, and contains the text "1 comment" to the far left of it. The right panel contains the comment thread view, which shows the ID of the thread at the top, the logged in user below that, as well as the box to enter comments, followed by the comment thread itself.

The comment functionality was built as a web application using Google App

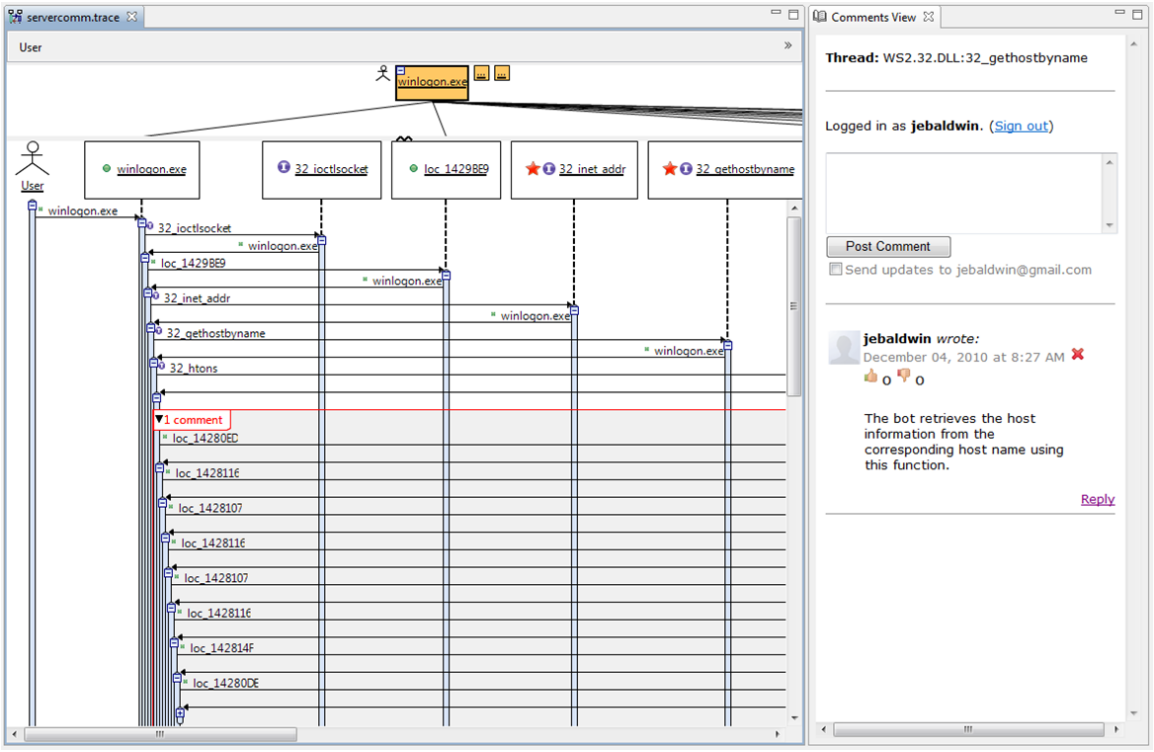

Figure 6.20: Tracks with Comment Threads.

Engine [112]. Comments are stored in the Google App Engine database with an ID of the artifact it pertains to. We then use this ID to query for the comment count, as well as a URL argument for loading the comment thread view. This ID includes a combination of the executable name, the source lifeline and parent names, the call lifeline and parent names, and the loop information if applicable. We also use a Google account login to add user information to each comment. This web application was incorporated into Eclipse as a view containing an embedded browser.

6.4 LegaSee: Visualiser Extension for Mainframe Assembly

In previous work of ours, the Visualiser provided by AJDT was found to be useful in looking at system modifications, introduced through patches, from a high-level in a tool called Eclippers[2]. In initial interviews and surveys with developers, only those in the mainframe group could see a potential application for a tool such as this

[2]Available at: `https://github.com/jebaldwin/Eclippers`

(Section 3.2.7). For example, trying to see where DSECTs are defined and used at a high level was quite difficult for a mainframe developer, who was using text search within the code to find all of its uses. Therefore we built a provider to show these specific HLASM constructs (DSECTs and CSECTs) from a system-wide perspective. We also provide navigation from the colored blocks within LegaSee to the listing files contained within the same project.

We previously showed the textual output from scraping listing files for data about DSECTs and CSECTs in Section 6.2. In order to use this data in LegaSee, we created a translator to generate the bars and stripes information, and created our own Visualiser provider. The file that provides information about the bars (or modules) within the system is called `content.vis`, while the file that provides data about the stripes is called `markup.vis`. A snippet from each file is shown in Figure 6.21. Here we see which group each bar belongs to but also the ID and size of that bar. We also see that the stripe refers to a specific bar, but has its own ID group, as well as an offset (where in the bar it begins) and a depth (length of the stripe).

```
Group:CBT019 Member:BLKSCAN Size:000D20 Tip:CBT019.BLKSCAN
Group:CBT019 Member:CA Size:0001C8 Tip:CBT019.CA
Group:CBT019 Member:CHECKPVT Size:0001C8 Tip:CBT019.CHECKPVT


Stripe:CBT019.BLKSCAN Kind:BLKSCAN Offset:00000 Depth:00D24
Stripe:CBT019.BLKSCAN Kind:IHADCB Offset:00000 Depth:00058
Stripe:CBT019.CA Kind:TFVSENV Offset:0 Depth:0
Stripe:CBT019.CA Kind:OPSAMD Offset:0 Depth:0
Stripe:CBT019.CA Kind:TFVSEOPE Offset:0 Depth:0
Stripe:CBT019.CA Kind:TFVSESR Offset:0 Depth:0
Stripe:CBT019.CA Kind:TSIDSYS Offset:0 Depth:0
Stripe:CBT019.CA Kind:TFVSERR Offset:0 Depth:0
Stripe:CBT019.CHECKPVT Kind:CHECKPVT Offset:00000 Depth:001D2
Stripe:CBT019.CHECKPVT Kind:CVT Offset:00000 Depth:00500
Stripe:CBT019.CHECKPVT Kind:CVTXTNT1 Offset:00000 Depth:0000C
Stripe:CBT019.CHECKPVT Kind:CVTVSTGX Offset:00000 Depth:00050
Stripe:CBT019.CHECKPVT Kind:CVTXTNT2 Offset:00000 Depth:00084
```

Figure 6.21: LegaSee Visualiser Files (content.vis and markup.vis).

The visualization of LegaSee for CBT019 is seen in Figure 6.22. The menu, which shows each of the CSECTs and DSECTs, is shown on the right of the screenshot. Each bar (or column) represents a module with its length equal to its last address, and each CSECT and DSECT is color-coded. CSECT and DSECT lengths are calculated by the difference between the start and end address (*length* in the XML). The start

locations of the CSECTs and DSECTs are equal to their start address. This view allows developers to see at a high-level where all of the DSECTs and CSECTs are located and also how much memory of the entire system they consume. Developers can also interact with the diagram by double-clicking on each colored segment, or module heading, to navigate to the listing files. There are also additional options provided by the AJDT Visualiser itself, such as zooming in and out, fit to view, limit view to affected bars, and group and member views (when packages are present).

Figure 6.22: LegaSee Visualization of CBT019.

This example is a memory view since lengths and locations are determined by addresses. A source view would instead mean that lengths and locations are based on lines of code. However, retrieving source lines for control sections is not as easy as it might seem. First, the source code does not have any line numbering, and second, the listing provides only the memory addresses. Statement line numbers for the beginning of CSECTs are provided, and present within the XML, though are currently only used for navigation within the textual listing file. A control section might be interrupted anywhere, and another control section started. Then the programmer might go back

to the first control section. Thus, a whole control section code might be composed from several segments that might be intertwined with other control sections. In summary, control section code might be scattered both in memory and source code.

Since neither a source code view, nor memory address view is a complete solution, either a combined view or two separate views will be required. Additionally, right now we see one large bar with many colored blocks. Therefore, it will be important to provide some ease of exploration by splitting the bars up first by module, then by subroutine, then into the CSECTs and DSECTs. We envision building an interactive treemap combined with bars and stripes to provide this interactivity to move from large to small granularity.

6.5 REwind: State Diagram Debugging Tool

The second overall most important issue from the requirements elicitation within the malware group, was being able to save debugging state (Section 4.3.4, requirement 2). We are currently working on a tool called *REwind* that will allow the user to start recording their actions within IDA Pro and save them periodically as states [113]. This means that the user can select a state within REwind and repeat all of their debugging actions up to that point within IDA Pro. REwind was created with the tedious nature of analyzing malware in mind—that is, once you have figured out how to run to one place in the execution, you can simply work from there without having to carefully repeat all of your previous steps.

An example of what the tool and such a state diagram would look like is shown in Figure 6.23, which shows states from running the Mariposa botnet. The first button on the REwind toolbar allows the user to begin recording a debugging session. This will ask you to choose which IDA Pro instance you want to use, as well as give a state name for that particular set of actions. We can see that the selected state is yellow, and we can choose to run to that state, delete that state or rename that state. We can also create connections between different states. If we have run to a particular state, then that state will be indicated as the current state by changing its border to a thick red line, and removing the yellow highlighting. States may also be freely moved around the canvas during use, however, the graph is rearranged by the Zest framework when reopened.

REwind works by saving actions for each state in the XML format shown in Figure 6.24. We can capture when each action is taken through the IDA Pro plugin

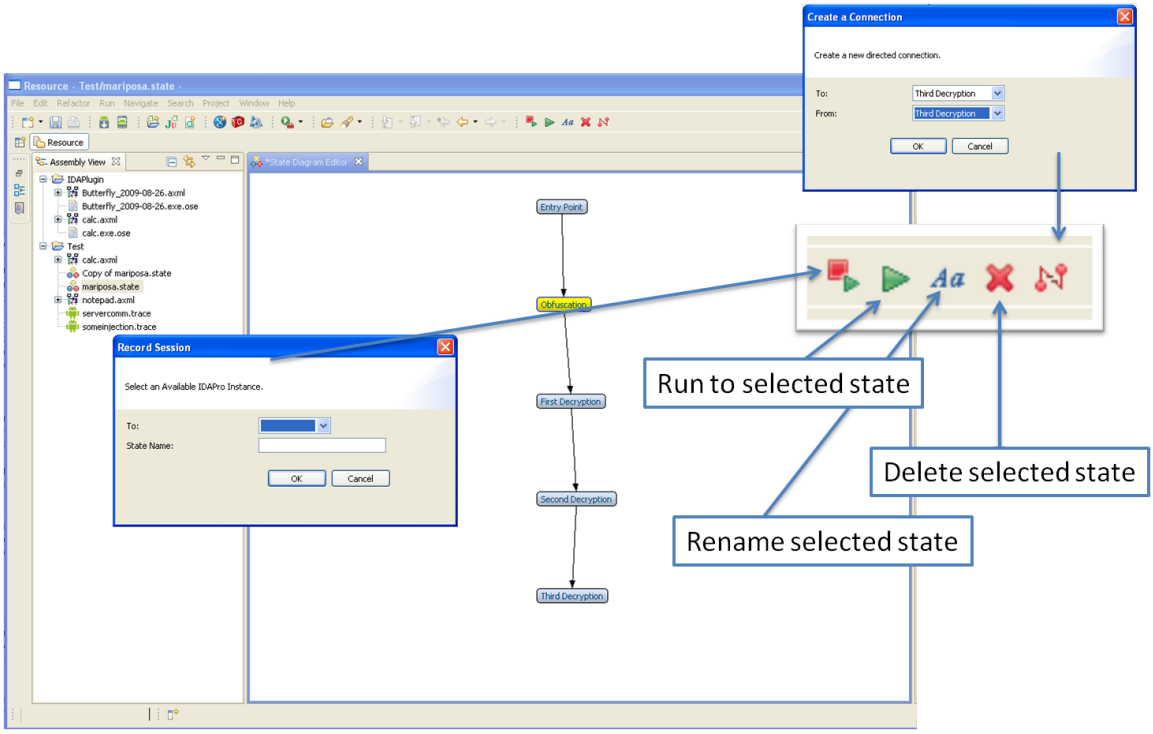

Figure 6.23: REwind Tool.

events we discussed previously.

This prototype works so far as to show that this idea is possible. However more work is needed to add all of the necessary actions to both the REwind plugin as well as the supporting IDA Pro plugin. This tool will be completed as part of future work.

6.6 Additional Contributions

We have discussed the contributions supported by AVA, however there were other contributions made by those working on different aspects of this project. While these are not directly related to the above contributions, they are related to the project itself and the requirements elicited. The first is support towards managing multiple IDA Pro instances. The second is comment support, while the third is related but includes tagging support for IDA Pro. Finally, the fourth is initial data flow support for IDA Pro.

```
<states>
   <state name=''Entry Point''>
      <action address=''41D469'' command=''set bp''/>
      <action address=''41D482'' command=''set bp''/>
   </state>
   <state name=''Obfuscation''>
      <action address='''' command=''runto''/>
   </state>
   <state name=''First Decryption''>
      <action address='''' command=''runto''/>
      <action address=''41D047'' command=''step''/>
   </state>
   <state name=''Second Decryption''>
      <action address='''' command=''runto''/>
      <action address=''41D047'' command=''step''/>
   </state>
   <state name=''Third Decryption''>
      <action address='''' command=''runto''/>
      <action address=''41D047'' command=''step''/>
   </state>
</states>
```

Figure 6.24: XML used by the REwind Tool.

6.6.1 Multiple Executables

The most important issue for the malware group was being able to disassemble more than one executable file at a time in IDA Pro. While Tracks itself does provide support for multiple executables by keeping tracking of which diagram relates to which instance of IDA Pro, there is no specific support within AVA for multiple executables. Rails [71] is a plugin developed for IDA Pro that aims to facilitate communication between multiple instances. Rails is a part of a related project called ICE or Integrated Comprehension Framework. While Rails does not address the exact elicited requirement (Section 4.3.4, requirement 1), which is to automatically launch a new IDA Pro from the current instance of IDA Pro, it does allow for comments to be propagated between instances, and eases navigation between instances.

6.6.2 Comment Support

Many of the issues brought up by the malware group had to do with comment support. In particular, saving comments during dynamic analysis, repeatable comments in external modules, and boilerplate comments. During the project, we had a student join to look at how IDA Python worked and to create some sample scripts to showcase this functionality. The student created two Python scripts that can be executed as

IDA Pro plugins to accomplish both viewing comments between modules, as well as comment templates. The former plugin supports comments between modules by storing comments in the network service, along with enough information to know where in the IDA project they were stored. The comments are then synced between instances of IDA Pro that have registered with the daemon. In the future, we would like to be able to have this run across multiple machines. The comment templates plugin allows a user to have a folder of comment header templates as text files, and easily add them from the IDA Pro plugin menu to the start of functions in the code. These plugins are freely available[3]. Figure 6.25 shows the flow of the comment templates feature, from text files containing the comment template to their addition to the plugins menu, to being inlined within the code. The requirements these plugins address are in Section 4.3.4, requirements 8 and 10.

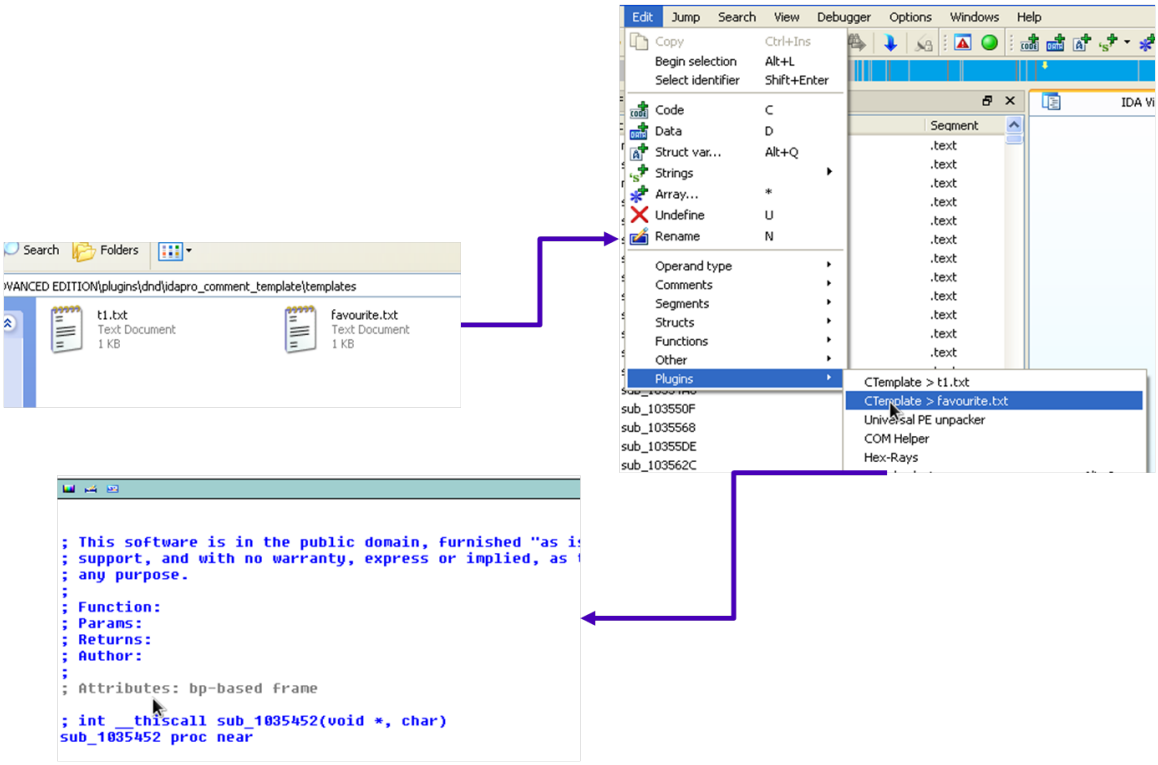

Figure 6.25: Comment Templates in IDA Pro.

[3]Available at: `https://github.com/cbenning/idapro_comment` and `https://github.com/cbenning/idapro_comment_template`

6.6.3 Tagging for IDA Pro

The fifth most important issue for the malware group was incorporating a tagging mechanism into IDA Pro. Another group at the University of Victoria (CHISEL) previously created an Eclipse plugin for tagging locations of interest called TagSEA [98]. This plugin worked for tagging Java, C and C++ source code, as well as breakpoints, tasks and resources. These tags would include keywords, data and author information, but also the ability to filter and navigate to tags. This functionality was already familiar to DRDC for use in Java projects, and was requested specifically for IDA Pro (Section 4.3.4, requirement 5). To this end, the initial version of TagSEA for IDA [114] was created, again by the CHISEL group. This port of TagSEA was built for IDA Pro as an IDA Python plugin and allowed tags to be added, jumped to, removed, filtered, and renamed—all within the IDA Pro user-interface.

Currently, another member of the CHISEL team is working on progressing TagSEA for IDA and has since renamed it to Tags for IDA Pro, so as not to be confused with the original TagSEA implementation for Java. This tool extends the initial version by adding a default author, and allowing additional columns to be created and associated with each tag. These columns can also be hidden or deleted as necessary. These comments are then saved to the IDA Pro Database (IDB) as opposed to an external file.

6.6.4 Data (Including Data Flow)

When speaking with assembly language developers about their challenges, data flow was the second most important topic brought up after control flow. Many of the requirements brought up by the malware group are in regard to data, including the flow of data through a program. While good tools exist for data flow such as *REF* (internal tool) and *Boomerang* [115], looking at IDA Pro only, *Sobek* seems to be the only attempt at solving data flow issues. Sobek is a deprecated IDA Pro plugin which provides simple data flow analysis and allows users to trace forward and backward [116]. When tracing forward, it shows instructions where the selected operand is used and/or propagated and tracing backward, it shows preceding instructions which the value of the selected operand depends on. Comments from the malware group on Sobek were: *it does not seem to support backward analysis above a few lines up, forward is a bit better but it gets confused easily, the interface is not clear, the addresses are all relative so it is difficult to follow*, and *some of it just does not make*

sense. We had issues even running Sobek with the newest version of IDA Pro. As a starting point, our student started to reimplement basics of the plugin as a Python script[4]. Currently, this script displays the register data at the point of a specific code segment.

6.7 Chapter Summary

This chapter has introduced the architecture overview of the AVA framework and each component within it. The framework consists of tools for control-flow, system-wide constructs and debugging. The framework also includes communication mechanisms with an example of an integrated industrial standard tool, in our case IDA Pro. Though we note that textual data can be fed directly into the tools if necessary.

Next, we discussed how data was obtained for both the mainframe and malware groups, in addition to the XML formats that are created from this data. Following this, we discussed each of our tools in-depth by providing descriptions and screenshots of their features, as well as how these features were implemented. These tools include Tracks, which uses sequence diagrams to show control flow and trace information, LegaSee which provides a high-level view of mainframe system constructs (CSECTs and DSECTs), and REwind, which provides malware analysts the ability to repeat debugging actions through a state diagram user-interface, therefore easing the process of reverse engineering.

We ended the chapter by discussing the contributions of others towards the malware requirements elicited in the work of this dissertation. In particular, we discussed working with multiple IDA Pro instances, extending comment support including the use of tags within IDA Pro, and how data flow can be supported through investigation of a deprecated IDA Pro plugin called Sobek.

This chapter has answered our third research question: *Can program comprehension tools for high-level languages be retrofitted to apply to low-level languages?* This encompasses the first portion of Phase III. This phase is concluded in the following chapter, which provides case studies using our proof of concept tools to further explore the disparities between groups.

[4]https://github.com/cbenning/idapro_dataflow

Chapter 7

Assessment of the AVA Project Lifecycle

This chapter provides an assessment of the AVA framework through case studies that aim to answer our final research question: *Are tools in our proof of concept framework effective at supporting the requirements of both groups?* We begin this chapter by providing our technical challenges and limitations. We then show how two tools that are part of AVA were unsuccessful at applying to issues within both groups, preventing the existence of a framework with universal application. However, we did have success with a universally applicable tool called Tracks, which was built to satisfy requirements for not only different groups of developers and therefore purposes, but also different assembly languages. We provide case studies using Tracks for the mainframe group using both control flow for Algol, as well as log data. Next we provide a case study of using Tracks in the malware context, analyzing the Mariposa botnet. These studies show that one tool *can* be built to work across both domains. Finally, we provide study limitations and threats to validity, and provide a discussion on the source of the divide between the two groups in our study.

7.1 Challenges and Limitations

This section discusses the challenges we faced for each group, as well as the limitations that existed due to the nature of the tasks performed within each group respectively.

7.1.1 Mainframe

There were three major issues for developing tools within this domain. These are the use of a mainframe in itself, issues with intellectual property, and the involvement of the stakeholders.

Mainframe Tool Development

With the malware group, we were able to easily write plugins for their primary reverse engineering tool. However this was not the case with the mainframe group. We did propose the idea to our CA contact early on and initial plans were to have research students on his end write these mainframe hooks. For us to write these hooks ourselves would have required an in-depth knowledge of the mainframe system, as well as access to it, both of which we did not have. Unfortunately when our contact left CA, we lost access to his graduate students that may have been able to complete this integration.

Intellectual Property Issues

While we use HLASM programs of similar size in our mainframe case studies, we did not have access to any of the programs actually created by CA. For this reason we could not test proof of concept tools with actual CA code, and use the results to discuss efficacy with developers.

Stakeholder Involvement

While we had close ties to the malware group, and recurring feedback on our implementation, we unfortunately did not have the same with CA. It is important to note that CA is a private company, and in particular we were looking at product development groups—not research groups. In such a case, it is hard to gain access to participants since they have hard deadlines to meet and therefore are less available time-wise. While we did obtain the time necessary for the requirements elicitation session, and a follow-up visit, any more time would have been hard to receive.

7.1.2 Malware

Within the malware domain, again there were three major issues. The first was associated with reverse engineering malware, the second was the lack of network connectivity for security reasons, and the third was proprietary issues.

Anti-Debugging and IDA Pro

Important limitations remain in regard to using Tracks with IDA Pro, most notably, anti-debugging traps. Anti-debugging traps are used to detect if a program is running under the control of a debugger and to prevent this runtime debugging [117]. Ideally, one would be able to run the executable with the sequence viewer open and afterwards investigate the entire call graph. Unfortunately, with the anti-debugging traps within Mariposa, this is an impossibility, since we cannot single step over the code. Therefore it is up to the user to know how to debug the executable. This means that they must have some prior knowledge about the system and cannot gain this prior knowledge directly from the sequence diagrams. We hope that better anti-anti-debugging tools will solve some of these issues. VERA addresses this through the Ether framework [118], which is an avenue that could be investigated.

Another limitation is the ability to properly trace stack locations while step tracing. Manually stepping through each instruction during a debugging session, we are able to log correctly since IDA Pro generates the names. However, when step tracing and executing code on the stack, the names are not generated.

Lack of Network Connectivity for Malware Analysis

The biggest challenge in this area is the common lack of network access on machines dedicated to malware analysis. In the case of Tracks' comments, using Google App Engine would no longer be possible offline. The comment service would need to change such that data would be saved locally and then shared after analysis by saving the file to an external drive manually, or from another machine with network access. This is also an issue for easily accessing API documentation, meaning that any documentation referenced would need to be stored locally.

Proprietary Issues

Since the DRDC is an operating government defence facility, there is much about the nature of their work we are not privy to. For example, we do not know the specific malware they reverse engineer, the exact nature of their work, or even whom their clients are. We only know what they are able to disclose to us. Observing them was also not an option, as it was with CA.

7.2 AVA: One Framework (Not to Rule Them All)

In this section, we examine two tools that are part of the AVA framework, that are not applicable to both groups: LegaSee and REwind.

7.2.1 LegaSee

The idea for LegaSee came during initial interviews with developers in the mainframe group, and witnessing their difficulty in locating CSECTs and DSECTs within the code. For example, using repeated text search functionality until the appropriate location of interest was found. We previously had experience in using the AJDT Visualiser to visualize system-wide constructs in other systems, and therefore thought it worth considering in this domain as well. As we have seen from our mainframe respondents in the exploratory survey, 10 of 25 respondents thought that it would be useful (some even commented very useful), 8 had no comment, 4 were unsure and only 3 said not useful. In contrast, only one malware respondent said it was useful with the remainder either seeing no use, or not understanding the concept. We therefore created the LegaSee tool as discussed in Section 6.4, for the mainframe group only. Due to the lack of interest from the malware group, and more specifically, lack of suggestion for any system-wide constructs, we did not attempt to create data or a case study for this particular tool. Table 7.1 summarizes the issues that LegaSee aims to address. This includes a high-level system view that displays the location and size of DSECTs and CSECTs, as well as navigation to them. There were no overlapping issues within the malware group.

Tool	**Requirement Category**	**Mainframe**	**Malware**
LegaSee	Browsing and Navigation	High-Level System View, Navigation to DSECTs and CSECTs	N/A
	Data	DSECT Module Location and Size	N/A
	References	CSECT Module Location and Size	N/A

Table 7.1: Summary of Requirements Supported by LegaSee.

7.2.2 REwind

The need for the REwind tool was made evident when we were trying to understand how to run the Mariposa botnet in IDA Pro in order to build malware-specific features into Tracks. The process was tedious, and error-prone. If any mistake was made, we either had to start over with analysis, or in the worst case, infected our machine and had to restore the system to begin again. This process in itself was also extremely time-consuming. The idea behind capturing a user's actions is a simple one, and also not a difficult one to implement using the IDA Pro API. This very request was also brought up during requirements elicitation at the malware group and was voted as the second most important issue with 300/400 points. While many issues related to debugging were brought up by the mainframe group, none were in regard to being able to repeat debugging commands. In contrast, the issues brought up by the mainframe group were very specific to their own work environment, and their work process likely did not follow the same repetitive nature of reverse engineering. Table 7.2 shows the issues that REwind addresses, namely: repeat debugging actions, iterative execution as a means of reverse engineering, avoiding anti-debugging traps, and state diagrams with descriptions. No corresponding issues are noted within the legacy group.

Tool	**Requirement Category**	**Mainframe**	**Malware**
REwind	Data	N/A	(Re-running with New Data)
	Debugging	N/A	Repeat Debugging Actions
	De-obfuscation	N/A	Discover Malware Intent through Iterative Execution, Anti-Debugging Trap Avoidance
	Documentation	N/A	State Diagram, State Descriptions
	Integration	N/A	IDA Pro

Table 7.2: Summary of Requirements Supported by REwind.

7.3 Tracks: One Tool to Rule Them All

This section provides case studies for Tracks in both the mainframe and malware contexts. In regard to mainframe, we look at an example of control flow for Algol, as well as how Tracks can be used to visualize log files. Within the malware context,

we look at how Tracks can be used with a particular piece of malware to visualize its activities. We further provide the results of interviews with malware analysts on the collaboration feature within Tracks which provides comment threads on specific points of control flow. This section shows that while these case studies may differ greatly, one tool can in fact be built to support both groups. These case studies show how Tracks' requirements shown in Table 7.3 are satisfied.

Tool	**Requirement Category**	**Mainframe**	**Malware**
Tracks	Browsing and Navigation	Navigation to Listing	Navigation within IDA Pro, Navigation History View
	Control Flow	Static Control Flow, Reversed Static, Trace Log	Static Control Flow, Reversed Static
	Debugging		Dynamic Control Flow
	De-obfuscation		Cycle/Loop Detection, API Call Patterns
	Documentation	Module Descriptions, Comment Threads, Save to Image File,	MSDN Documentation, Comment Threads, Save to Image File
	Integration	(Socket Message Capable)	IDA Pro
	References		DLLs Referenced

Table 7.3: Summary of Requirements Supported by Tracks.

7.3.1 Mainframe: Static Control Flow for Algol

The ability to ascertain control flow for mainframe systems has been anecdotally referred to as the "Holy Grail". As we mentioned previously in Section 6.2.1, one of the major challenges with HLASM code is that it contains no explicit concept of subroutines. We therefore rely on coding conventions and known patterns to infer where a subroutine call takes place, and where a subroutine exists. In this section we show how Tracks can be used to show control flow for mainframe systems, specifically the Algol package [105]. While the XML format is the same for Algol as for the CBT019 example, we show the static control flow data in Figure 7.1.

We previously showed how Tracks presents the list of functions available in an executable in a tree view. The tree view used for HLASM systems is similar, however this tree view uses a nested structure. This is shown in Figure 7.2. The top-level items represent the modules within the package, which are labelled as <section>

```
<softwarePackage name=''Algol''>
  <copybook name=''FSACONV''>
  </copybook>
  <copybook name=''FSAREA''>
  </copybook>
  <copybook name=''IEX60000''>
  </copybook>
  <copybook name=''WORKAREA''>
  </copybook>
  <section name=''CA''>
      <functionEntryPoint address=''0'' section=''TFVSENV'' name=''TFVSENV''
          index=''0''/>
      <functionEntryPoint address=''0'' section=''OPSAMD'' name=''OPSAMD''
          index=''0''/>
      <functionEntryPoint address=''0'' section=''TFVSEOPE'' name=''TFVSEOPE''
          index=''0''/>
      <functionEntryPoint address=''0'' section=''TFVSESR'' name=''TFVSESR''
          index=''0''/>
      <functionEntryPoint address=''0'' section=''TSIDSYS'' name=''TSIDSYS''
          index=''0''/>
      <functionEntryPoint address=''0'' section=''TFVSERR'' name=''TFVSERR''
          index=''0''/>
  </section>
  <section name=''IBM''>
      <functionEntryPoint address=''0'' section=''IJBPROC'' name=''IJBPROC''
          index=''0''/>
  </section>
  <section name=''IEX00''>
      <functionEntryPoint address=''00000'' section=''IEX00000'' name=''IEX00000''
          index=''0052C''/>
   </section>
  <section name=''IEX10''>
      <functionEntryPoint address=''00B70'' section=''IEX10001'' name=''IEX10001''
          index=''00016''>
         <function address=''000B70'' name=''IEX10001'' index=''1''
             section=''IEX10001''>
            <call calladdress=''000EE8'' name=''COBSPEC'' functionaddress=''000EE8''
                index=''external''
                  externalFile=''IEX11001''/>
         </function>
      </functionEntryPoint>
      <functionEntryPoint address=''00000'' section=''IEX10000'' name=''IEX10000''
          index=''00B6C''>
         <function address=''00041C'' name=''TSTDCBRT'' index=''1''
             section=''IEX10000''>
            <call calladdress=''0004AE'' name=''COMEXRT'' functionaddress=''0004AE''
                index=''local''
                  externalFile=''IEX10000''/>
         </function>
         <function address=''0004AE'' name=''COMEXRT'' index=''2''
             section=''IEX10000''>
            <call calladdress=''000526'' name=''CLOSE'' functionaddress=''000526''
                index=''local''
                  externalFile=''IEX10000''/>
         </function>
         <function address=''000526'' name=''CLOSE'' index=''3''
             section=''IEX10000''/>
      </functionEntryPoint>
  </section>
```

Figure 7.1: Static Control Flow Snippet for Algol.

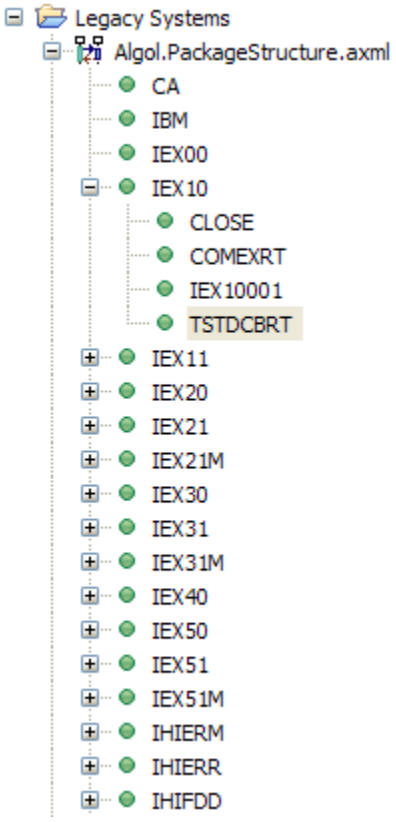

Figure 7.2: Nested Tree View of Modules and Subroutines in Algol.

elements within the XML data. These modules then contain subroutines, as defined and ascertained by the python scripts discussed in Section 6.2.1.

Once we have selected a subroutine from the second level of the tree, we can view the same static control flow information as we have previously seen for the calculator example. The static control flow information for the `TSTDCBRT` function is shown in Figure 7.3. In this example, we see that the package diagram at the top is used to show us from which module this subroutine comes, as opposed to which executable or DLL.

7.3.2 Mainframe: Log File Visualization

The previous example showed the expected usage for viewing control flow information for an HLASM package. However, during our requirements elicitation session with the mainframe group, we ended with two one-hour long sessions to discuss issues brought up with two developers to ensure we understood the requirements. During one of these sessions, the developer showed us a log file (or trace table) that he used to investigate

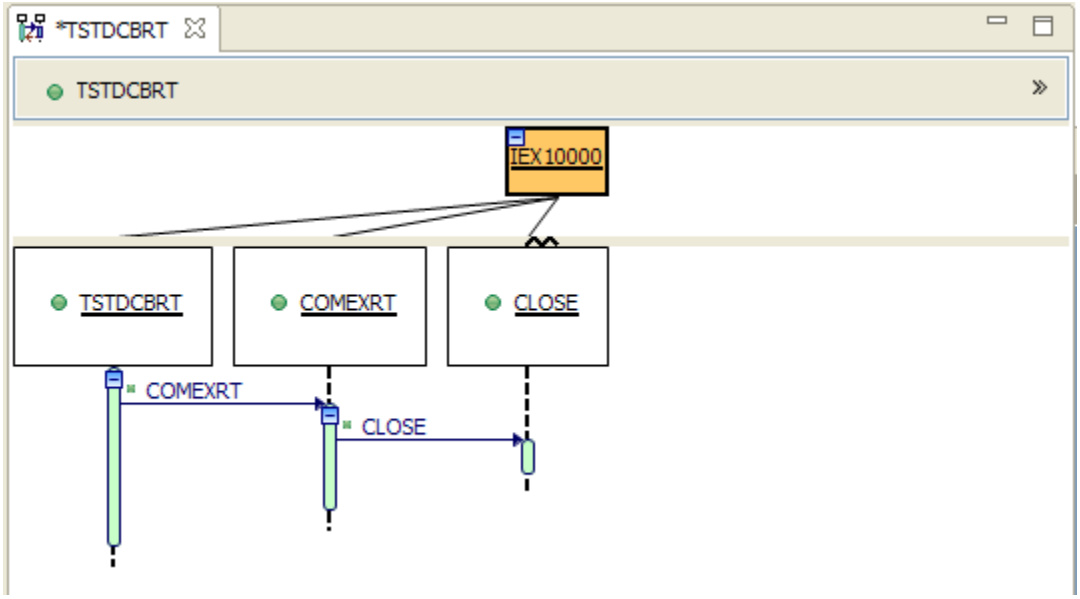

Figure 7.3: Control Flow of TSTDCBRT Function in Algol.

issues. It was immediately evident that such a log file could be visualized within the Tracks tool. The developer told us that the log file could easily be downloaded via FTP from the mainframe. After this visit, we worked with this developer over email to further refine the design of this specific sequence diagram. A snippet from this log file is shown in Figure 7.4. The entries for XPB WAIT indicate that the thread is suspended and the next bit of work will start with an XPB DISPATCH. If the XPB address is the same then it is the same thread that is resuming. In this way, we do have the data necessary to provide some sort of multi-threading diagram support within Tracks, however we have not implemented any such support at this time.

The $NMXCTL entries are when a module switches control to another module, instead of calling and expecting a return. In this trace NM000038 eventually does a $NMXCTL to NM000040. NM000040 then does a $NMXCTL into NM000038. What is actually happening is that NM000040 is a module that does setup for running the NCL language (NM000038). Therefore, this sequence is an NCL procedure calling another NCL procedure. The developer did not expect that the trace tool would be able to deal with this, but would expect it to simply not create issues for the tool. We therefore do not recognize calls of that kind in Tracks.

Another issue with this particular log trace is that there are specific calls that are not logged by the system. For example, NM000233 is the variable manager for their internal scripting language (NCL). It is set not to trace because otherwise

```
  RETURN TO:    NM000038 FROM NM000241 AT 9BE2D6DA         REGISTER 15 0000000C             31
       03F0F3F8F2F4F1029BE2D6DA0000000C

  CALL TO:      NM00025C FROM NM000038 AT OFFSET +0BBA     CALL ADDR   9BE2D6DA F/C 00      31
       02F2F5C3F0F3F8029BE2D6DA0BBA0000

  CALL TO:      NM000103 FROM NM00025C AT OFFSET +00F8     CALL ADDR   9BE625A0 F/C 44      31
       02F1F0F3F2F5C3029BE625A000F84404

  RETURN TO:    NM00025C FROM NM000103 AT 9BE625A0         REGISTER 15 00000000             31
       03F2F5C3F1F0F3029BE625A000000000

  CALL TO:      NM000276 FROM NM00025C AT OFFSET +0E40     CALL ADDR   9BE632E8             31
       02F2F7F6F2F5C3009BE632E80E406B04

XPB WAIT:       NM000276 AT OFFSET +00B4                   XPB ADDRESS           1CD8D2F8   24
     06F2F7F61CD8D2F8800510F400B40000

$NMPOST:        ISSUED BY UNKNOWN MODULE AT 800E875E       NMPOST CODE 00000000             24
     07FFFFFF800E875E000E663400000000
                                                           TARGET ECB  000E6634
$NMPOST:        ISSUED BY UNKNOWN MODULE AT 800E758E       NMPOST CODE 00000000             24
     07FFFFFF800E758E1E88DF2400000000
                                                           TARGET ECB  1E88DF24
XPB DISPATCH:   NM00029I AT OFFSET +00A6                   XPB ADDRESS           1DF492F8   24
     01F2F9C91DF492F8800E76F600A60100

  RETURN TO:    NM000090 FROM NM00029I AT 8004BBAC         REGISTER 15 00000000             31
       03F0F9F0F2F9C9608004BBAC00000000

  CALL TO:      NM000065 FROM NM000090 AT OFFSET +0314     CALL ADDR   8004BA4C F/C 0C      31
       02F0F6F5F0F9F0028004BA4C03140C00

  RETURN TO:    NM000090 FROM NM000065 AT 8004BA4C         REGISTER 15 00000000             31
       03F0F9F0F0F6F5028004BA4C00000000

  $NMPOST:      NM000090 AT OFFSET +032A                   NMPOST CODE           00000000
       07F0F9F0032AF2C41C7E99F800000000
                                                           TARGET ECB  1C7E99F8
  $NMPOST:      NM000090 AT OFFSET +036E                   XPB ADDRESS           1D2816A4
       07F0F9F0036EF402FFFFFFFF1D2816A4
                                                           THIS IS AN XPB POST
THREAD FINISH:  NM000090                                   TERMINATED XPB        1DF492F8
     09F0F9F01DF492F89BEA83F61DB93604

XPB DISPATCH:   NM000042 AT OFFSET +0230                   XPB ADDRESS           1DA543B4   24
     01F0F4F21DA543B4800E854002300100

  CALL TO:      NM000035 FROM NM000042 AT OFFSET +052C     CALL ADDR   800E883C F/C 18      31
       02F0F3F5F0F4F202800E883C052C1804

  RETURN TO:    NM000042 FROM NM000035 AT 800E883C         REGISTER 15 00000000             31
       03F0F4F2F0F3F502800E883C00000000

  $NMXCTL TO:   NM000074 FROM NM000042 AT OFFSET +03E0     XCTL ADDR   800E86F0             31
       0AF0F7F4F0F4F202800E86F003E08000

  CALL TO:      NM000045 FROM NM000074 AT OFFSET +02BE     CALL ADDR   9BEA83F6 F/C 0C      31
       02F0F4F5F0F7F4029BEA83F602BE0C04

  RETURN TO:    NM000074 FROM NM000045 AT 9BEA83F6         REGISTER 15 1DDA6148             31
       03F0F7F4F0F4F5029BEA83F61DDA6148

  CALL TO:      NM000043 FROM NM000074 AT OFFSET +0306     CALL ADDR   9BEA843E F/C 00      31
       02F0F4F3F0F7F4029BEA843E03060004

  RETURN TO:    NM000074 FROM NM000043 AT 9BEA843E         REGISTER 15 00000004             31
       03F0F7F4F0F4F3B49BEA843E00000004

  CALL TO:      NM000065 FROM NM000074 AT OFFSET +035E     CALL ADDR   9BEA8496 F/C 0C      31
       02F0F6F5F0F7F4029BEA8496035E0C00

  RETURN TO:    NM000074 FROM NM000065 AT 9BEA8496         REGISTER 15 00000000             31
       03F0F7F4F0F6F5029BEA849600000000
```

Figure 7.4: Mainframe Log File.

every variable reference would be traced and create too much output. The developer thought the best solution would be to alias NM000233 to NM000038. This is shown in Figure 7.5. The issue being that we never know how we arrived at NM000233 so including it creates an erroneous diagram.

```
CALL TO:       NM00023C FROM NM000036 AT OFFSET +08DC      CALL ADDR    9BE41D7C F/C 6B
         31 02F2F3C3F0F3F6029BE41D7C08DC6B00

RETURN TO:     NM000036 FROM NM00023C AT 9BE41D7C          REGISTER 15 00000000
                 31 03F0F3F6F2F3C3029BE41D7C00000000

RETURN TO:     NM000038 FROM NM000036 AT 9BE2D9A2          REGISTER 15 00000004
                 31 03F0F3F8F0F3F6029BE2D9A200000004

CALL TO:       NM000234 FROM NM000233 AT OFFSET +0D24      CALL ADDR    9BE2F7EC F/C A0
         31 02F2F3F4F2F3F3029BE2F7EC0D24A000
```

Figure 7.5: Mainframe Log File with Calls to NM000233.

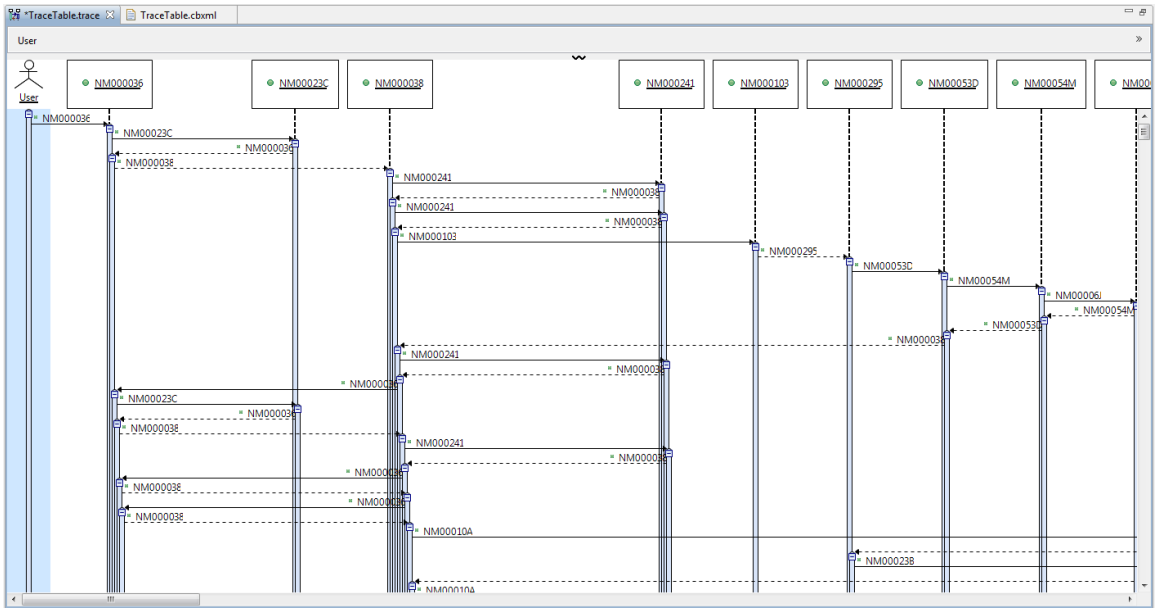

Figure 7.6: Tracks for Log File.

Figure 7.6 shows Tracks visualizing this trace file by ignoring all calls to and from NM000233. After discussion with the developer, he mentions that associating module descriptions with the module names of the activations would be useful. He provided a flat file dump of the module names and descriptions from the mainframe. Figure 7.7 shows how this list appears within the mainframe when their TITLE value is set,

while Figure 7.8 shows the description available when a particular module is viewed within the Source Management System. Figure 7.9 shows the flat file containing short descriptions for the module. Since Tracks includes module information for every activation, this provides an area where description information could be contained.

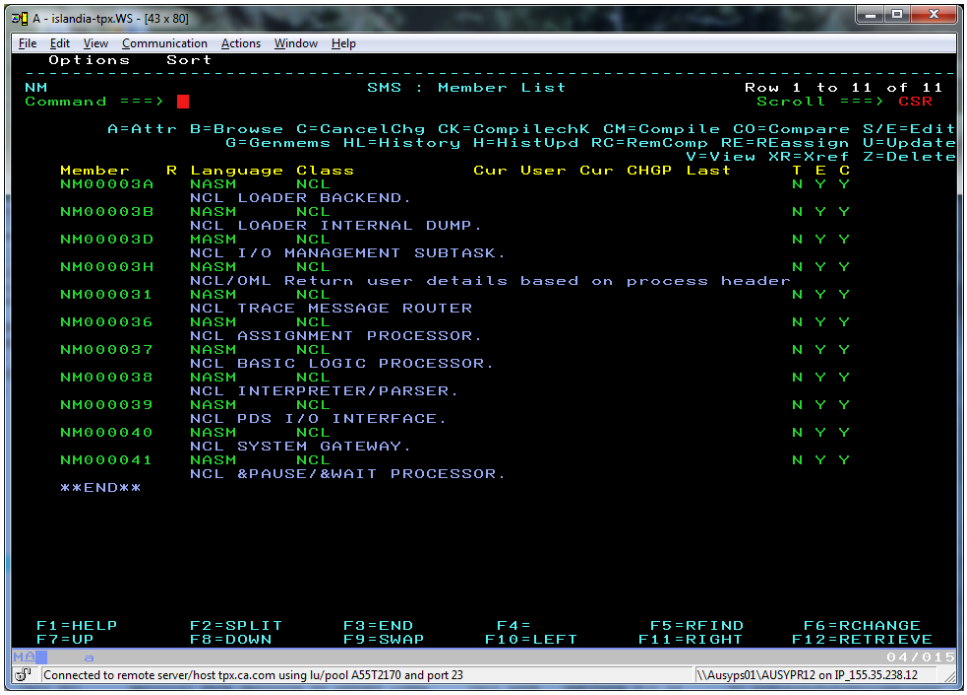

Figure 7.7: Listing of Modules with Title Values Set.

7.3.3 Malware: Mariposa Botnet Case Study

This section provides a case study of using Tracks in the malware context, and for analyzing malware itself. Previously we saw how Tracks could be used for a regular executable (calc.exe). However in this section we see how Tracks can truly be leveraged in the analysis of malware, which is purposefully obfuscated to hide its true malicious intent. We first discuss what Mariposa is and then show how Tracks can be used to visualize different phases of the botnet.

About Mariposa

Mariposa is a botnet, a collection of computers under the control of a single malicious entity. The most dangerous capability of this botnet is that it can download and execute arbitrary programs, which means the bot master can infinitely extend the

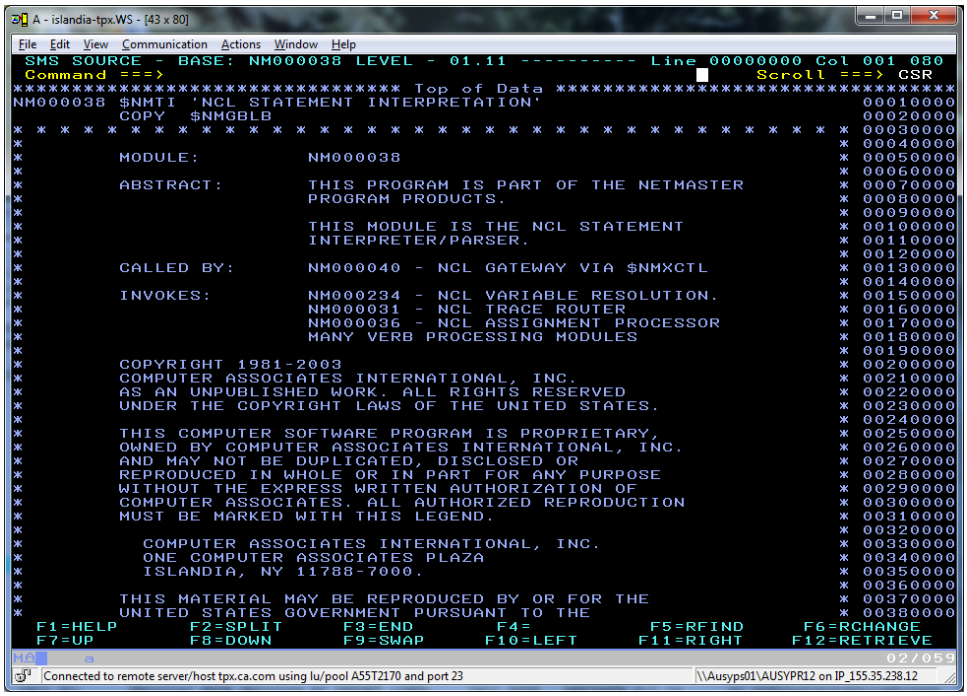

Figure 7.8: Description Field in the Source Management System.

```
NM000A00 apinm - bootstrap loader
NM000A10 loaded apinm front-runner module
NM000A11 apinm - init anwa and read ctl file
NM000A13 apinm - handle misc local requests
NM000A14 apinm - send event trigger to pa event handler
NM000A19 apinm - loaded mod - general subroutines
NM000A20 rexx-callable ('address link NMREXAPI') pgm to init apinm
NM000A40 PASS FEATURE INITIALIZATION
NM000A41 NCL &PASSCHK PROCESSING VERB
NM000A43 BLACK BOX PASS MODULE
NM000BSE NetMaster SYSDB Batch Utility: ESTAE/Dump services
NM000BSG NetMaster SYSDB Batch Utility: GVT
NM000BSH NetMaster SYSDB Batch Utility: Commands definitions table
NM000BSK SYSDB Filter compiler variable table
NM000BST NetMaster SYSDB Batch Utility: Tables
NM000BSX NetMaster SYSDB Batch Utility: GREXX interface init
NM000BSY NetMaster SYSDB Batch Utility: GREXX interface GVT
NM000BS0 NetMaster SYSDB Batch Utility: startup module
NM000BS1 NetMaster SYSDB Batch Utility: command input
NM000BS2 NetMaster SYSDB Batch Utility: command parse
NM000B20 EXTERNAL OBJECT SERVICES API VIA PPI ASM3
NM000B21 EXTERNAL OBJECT SERVICES API VIA PPI ASM3
NM000B22 EXTERNAL OBJECT SERVICES API VIA PPI ASM3
NM000B23 EXTERNAL OBJECT SERVICES API VIA PPI ASM3
NM000B24 EXTERNAL OBJECT SERVICES API VIA PPI ASM3
```

Figure 7.9: Mainframe Module Description Flat File.

functionality of the malicious software. This also reduces or eliminates the detection rates of traditional host detection methods. Due to this capability, 1500 variants of Mariposa have been detected so far and an estimated 12.7 million computer systems have been compromised (circa October 2009) [111]. We used information from [111] and [6] to create the following three use cases, which show meaningful interactions through sequence diagrams.

The botnet has three phases: obfuscation, decryption and injection. The obfuscation phase hides the intended functionality of the code. The decryption phase decrypts code and data that is used in the injection phase. The final phase, injection, is where Mariposa injects code into a legitimate system process in order to compromise the operating system. We now look at some of these particular interactions in detail and show how Tracks' sequence diagrams can aid in identifying these interaction patterns. There are two particular areas of interest as we will see: the detection of loops/cycles, and the patterns of system calls as a means to detect malicious intent.

We use the Mariposa variant with the MD5 hash of `3E3F7D8873985DE888CE320092ED99C5`. The instructions for running this variant of Mariposa on Windows XP, in order to create these control flow diagrams with Tracks, is shown in Appendix E.

Obfuscation

Obfuscation is the art of transforming *the application into one that is functionally identical to the original but which is much more difficult... to understand* [99]. Commonplace within malware, the Mariposa bot includes one such obfuscation, a large cycle that does nothing but useless computations, 889,976,605 times! The purpose is to confuse the person who is debugging the software and also, possibly, to confuse automated unpacking tools or malware analysts. This obfuscation shows up as a cycle in Tracks, discussed in more detail in the injection phase. After this loop, control flow is transferred to an address pushed onto the stack, which begins the decryption phase.

Decryption

After the obfuscation phase, Mariposa must decrypt its own actual payload code. The first decryption layer XORs[1] the range of data between addresses `0x41D000` and `0x41D4C0` with the constant `0x0CA1A51E5`. The address `0x41D047` is then pushed to

[1] An XOR is a logical bitwise exclusive OR.

the stack as a return value to transfer control flow to this address at function return. The assembly code for this first decryption layer is shown in Figure 7.10 and the corresponding sequence diagram is shown in Figure 7.11. Remember that large loops occurring within a single function are colored red to show that a large loop occurs, which may be suspicious.

```
loc_13FFA6:
   xor     dword ptr [ecx], 0CA1A51E5h
   nop
   add     ecx, 4
   nop
   nop
   cmp     ecx, offset dword_41D4C0
   jl      short loc_13FFA6
   nop
   push    offset loc_41D047
   retn
```

Figure 7.10: Decryption Loop in x86.

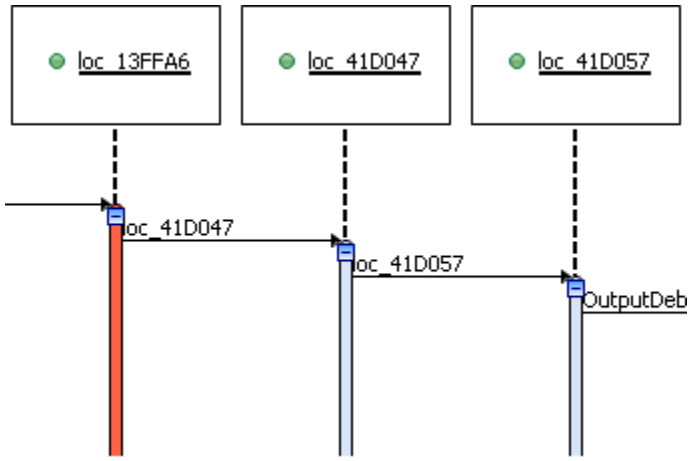

Figure 7.11: Decryption Loop in the Sequence Viewer.

Injection

There has been much work in the malware field to detect intrusion through sequences of system calls [119, 120]. The idea being that short sequences of system calls executed by running processes can be a good discriminator between what is a normal process and what is an abnormal one. For example, finding code that modifies the registry that is not an installer or code that injects bytecode into another process

could indicate malicious activity. Such code injection is a technique to hide malicious processes within a legitimate process and is a popular method for compromising an operating system. We hope that this type of information can be discovered more easily through the sequence viewer. We now consider the system call patterns involved in Mariposa's injection process and their functionality once infected.

Preparing for Injection: The first step of injection is to prepare the data that is used by the injected code. This includes the creation of directories, getting the operating system version (need to check if `CreateRemoteThread` can be called) and creating files. The next step is to find the process to inject into. To do so, `CreateToolhelp32Snapshot` is called to obtain a snapshot of the processes that are running in the system. `Process32First` is called to retrieve the first of these processes and then `Process32Next` is called until the required process name is found. There are 107 total lines of code for this step. Figure 7.12 shows the snippet of this code which calls `Process32Next`. As we can see, the call is made on Line 8. However, since the address is stored in the register, it is not easily apparent statically that this is a call to `Process32Next`. This information becomes apparent while debugging and stepping through the call. It is even more apparent in the sequence diagram, as shown in Figure 7.13.

```
loc_13591F:
   lea     ecx, [ebp+var_128]
   push    ecx
   mov     edx, [ebp+var_134]
   push    edx
   mov     eax, [ebp+var_12C]
   mov     ecx, [eax+6Ch]
   call    ecx
   test    eax, eax
   jnz     loc_135899
```

Figure 7.12: Finding Each Process in x86.

Injection: Having found the process to inject, Mariposa calls `OpenProcess` to get a handle to this process. It then calls `VirtualAllocEx` to allocate memory within the target process, `NtWriteVirtualMemory` to inject the code, followed by `CreateRemoteThread` to execute it.

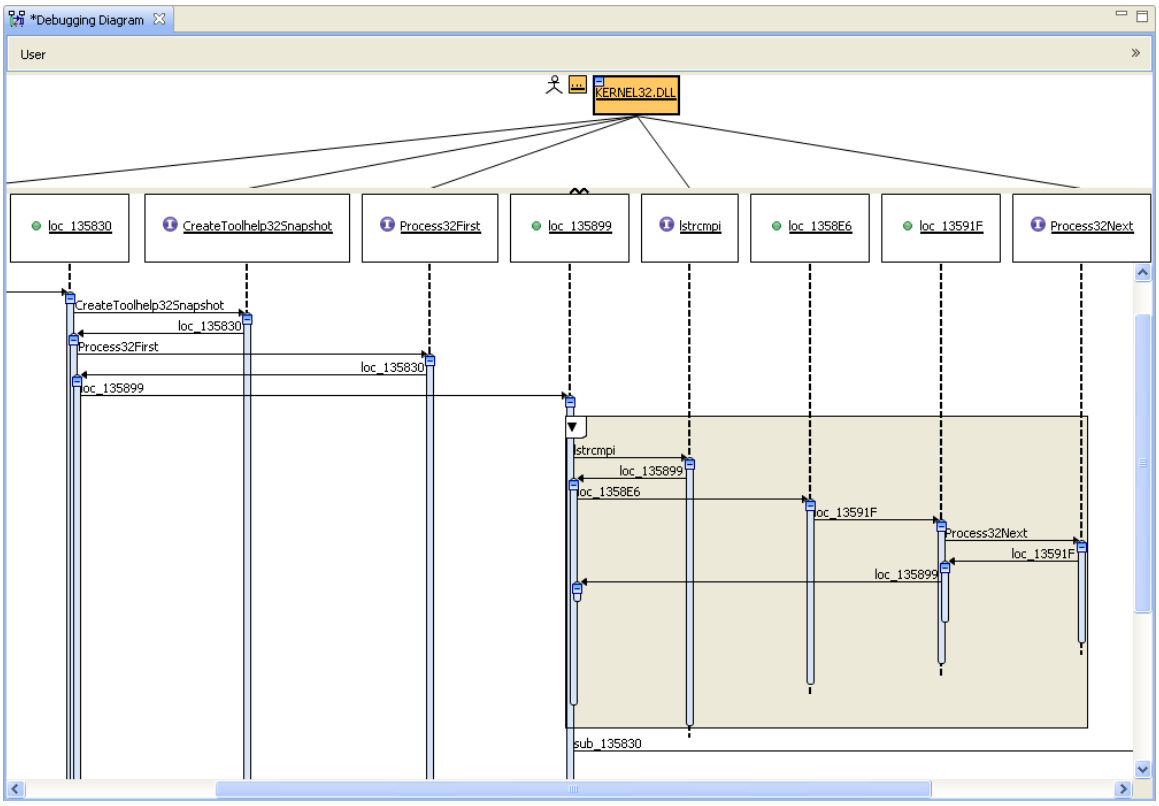

Figure 7.13: Finding the Process to Inject.

Injected Process: Once the process has been injected, we can follow the control flow from the address where the injection occurred. In this study, we have actually injected `winlogon.exe` instead of `explorer.exe`. We did so because injecting into `explorer.exe` makes working on the computer extremely difficult. We used `winlogon.exe` because it is also a system process (which Mariposa checks for) and has the same name length, making it easy to modify during runtime.

Several interesting things occur within the injected process, as seen in Figure 7.14. These include ensuring that the files to reinfect the system on startup are present, that their values are set in the registry, and that communication with the server is set. Here we only discuss this last step.

The infected process connects to the command and control server. The events that take place here are creating a socket with `32_ioctlsocket` and then `32_inet_addr` converts the domain names into proper addresses. Next, the bot retrieves host information from the host name by calling `32_gethostbyname`. The `32_htons` function converts an unsigned short number from the host to a TCP/IP network byte order (big endian). In order to authenticate with the server, an encrypted magic word is

sent. Figure 7.14 shows a collapsed loop at this point in the diagram. This loop is responsible for encrypting the magic word. Once the magic word is encrypted, it is sent using the `32_sendto` function. It then receives a reply from the server using the `32_recvfrom` function (not shown). The injected process is then ready to decrypt and decode received commands.

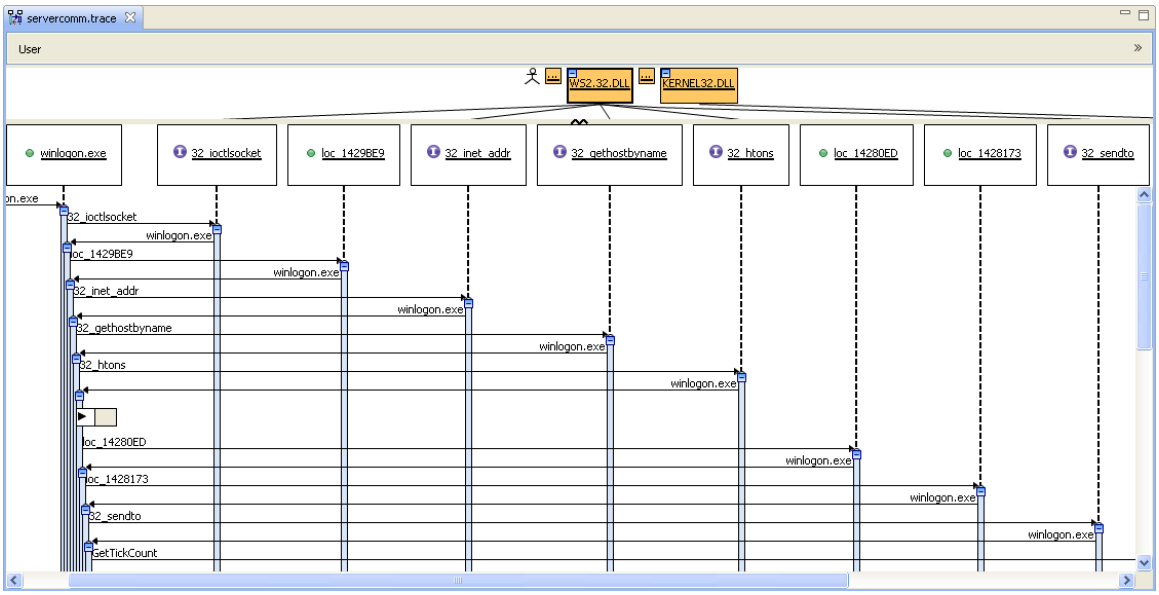

Figure 7.14: Communication with the Server.

Is Value Added by Tracks?

Table 7.4 compares the ways in which IDA Pro and Tracks address the control flow requirements from our exploratory survey. Relative to IDA Pro, the visualization in Tracks reduces cognitive overhead by better supporting navigation, showing a more intuitive control flow and allowing zoom/collapse interaction with visual cues. Additionally, added features integrated into this visual framework—system call patterns, dynamic traces, loops and recursion, call ordering and traces involving more than one executable. The data is available to compare traces and analyze branch frequency in Tracks, whereas no data is easily available for export directly from IDA Pro.

7.3.4 Malware: Collaboration and Documentation in Tracks

This section shows how features can be added to Tracks to support additional functionality, seamlessly within the same tool. We previously discussed how the comment

Control Flow Issue	IDA Pro	Tracks
API Call Pattern	Static control flow with local functions only. No search or navigation capability.	Static or dynamic control flow with both local and external functions. Functions can be located through a tree view and customized perspective.
Loops and Recursion	No call ordering and no indication if call is made more than once	Shows the order of calls, including each time a call is made. Also shows recursion, loops and cycles.
Trace Comparison	No support.	No support. Data is available.
Data Required to Reach Execution Points	No support.	No support.
Branch Frequency	No support.	No support. Data is available.
Multi-Executable Traces	No support.	Can merge call paths into one Tracks diagram from multiple IDA Pro instances.

Table 7.4: Comparison of IDA Pro and Tracks.

threads were implemented alongside Tracks in Section 6.3.5. In order to gauge how useful our approach was, we contacted five survey participants who gave us permission to do so. Of those five, we conducted telephone interviews with three for approximately 30 minutes each. They were issued a demo video of the features in advance[2]. Participants were able to freely respond with comments but some of the questions we asked included:

- Is there anything particularly useful in the tool?
- Is there anything not useful?
- Are there any features missing?
- Do you think the stars/colors are the best representation?
- Are there other artifacts you would like to be able to add comments to?
- Do you think there is a need to create documentation, for personal use, in the same fashion?
- How many comments do you foresee being in a thread?

The three interviews are summarized below, followed by a brief overview and analysis of these results.

[2]The demo video is available at `http://jenniferbaldwin.info/ava/CommentDemo.mp4`

Interview 1: Comments on Basic Blocks

Rob[3] typically does not work on projects with other people but his clients do. He believes that reverse engineering tools fall down without collaboration but also remarks that this is changing as more tools are adding support.

As for usefulness, he finds that the history of comments is useful and that it is similar to having a WIKI alongside the diagram. He also likes the ability to see comments by their post date, but would want to be able to navigate from the newest comment to where it pertains to in the code.

Extra features he would like to see include being able to incorporate comment threads directly into a report either by exporting it or printing it to PDF. He often puts snippets of code into reports alongside the documentation, which he would like to have happen automatically. Rob would also want to be able to look at the differences in comments between two executables, for example if a new version of an executable is released. He would also like to be able to link comment threads, for example if code is duplicated in another executable, and to have the comments themselves link to the Internet or to files. Finally, Rob could see the potential to have comments approved, for example by a team leader. This means that if a particular part of the code has been approved by reaching a level of saturation of comments, then no more analysis would need to occur on it. In this way it becomes a tool for managing work flow.

As for other artifacts, he often comments basic blocks of code, for example, *if statements* and would like to be able to add comments to these. One idea for this would be to add boxes to the lifeline image that could be double-clicked to load their comment threads.

Rob mentioned that having the ability to create private documentation (documentation for personal use) would be useful as a personal workspace for rough notes.

As for the length of comment threads, this was difficult to answer but Rob suggested having a tree hierarchy or nested overlay for long comment threads and thought that interesting sections might have 8 to 12+ comments.

Interview 2: Comments on Mouseover

Joe usually works as an individual, with maybe one other person at a time. However, he does not work in a group so he is not used to working in this context and has a greater need to create documentation privately. He usually creates documentation

[3]The names used in this study are fictitious.

with the comments in IDA Pro, but does find the ability to ask questions to other people interesting.

Joe believes that double clicking to have the comments load on the right is disruptive to the work flow and instead suggests that they appear on mouseover with a summary of comments in place. In this way, the user does not have to navigate to the side panel and back again, wondering what they were looking at, especially in a large trace.

He would also like to be able to pull in information from other sources. For example, he is almost always looking up functions on the Microsoft Developer Network (MSDN) since he forgets what the parameters are. The first thing he would like to do, if he had the prototype code, is to add links to it from the comments. There is an IDA Pro plugin that provides similar functionality [121]. A screenshot of this tool having imported functions is shown in Figure 7.15. Here we can see that MSDN documentation describing function parameters has been pulled in as comments into the code. However, since speaking with Joe, we have added a view alongside Tracks to show the MSDN documentation (discussed in Section 6.3.4).

```
.text:77E344C0 ; FUNCTION CHUNK AT .text:77E0C04E SIZE 00000006 BYTES
.text:77E344C0 ; FUNCTION CHUNK AT .text:77E1122F SIZE 00000008 BYTES
.text:77E344C0
.text:77E344C0                 mov     edi, edi
.text:77E344C2                 push    ebp
.text:77E344C3                 mov     ebp, esp
.text:77E344C5                 xor     eax, eax
.text:77E344C7                 cmp     [ebp+bManualReset], eax ;  If this parameter is TRUE, the function creates a manual-reset event
.text:77E344C7                                         ; object, which requires the use of the ResetEvent function to set the
.text:77E344C7                                         ; event state to nonsignaled. If this parameter is FALSE, the function
.text:77E344C7                                         ; creates an auto-reset event object, and system automatically resets
.text:77E344C7                                         ; the event state to nonsignaled after a single waiting thread has been
.text:77E344C7                                         ; released.
.text:77E344CA                 jnz     loc_77E0C04E
.text:77E344D0
.text:77E344D0 loc_77E344D0:                           ; CODE XREF: CreateEventA(x,x,x,x)-28471↑j
.text:77E344D0                 cmp     [ebp+bInitialState], 0 ;  If this parameter is TRUE, the initial state of the event object is
.text:77E344D0                                         ; signaled; otherwise, it is nonsignaled.
.text:77E344D4                 jnz     loc_77E1122F
.text:77E344DA
.text:77E344DA loc_77E344DA:                           ; CODE XREF: CreateEventA(x,x,x,x)-2328E↑j
.text:77E344DA                 push    1F0003h
.text:77E344DF                 push    eax
.text:77E344E0                 push    [ebp+lpName]    ; The name of the event object. The name is limited to MAX_PATH
.text:77E344E0                                         ; characters. Name comparison is case sensitive. If lpName matches the
.text:77E344E0                                         ; name of an existing named event object, this function requests the
.text:77E344E0                                         ; EVENT_ALL_ACCESS access right. In this case, the bManualReset and
.text:77E344E0                                         ; bInitialState parameters are ignored because they have already been
.text:77E344E0                                         ; set by the creating process. If the lpEventAttributes parameter is not
.text:77E344E0                                         ; NULL, it determines whether the handle can be inherited, but its
.text:77E344E0                                         ; security-descriptor member is ignored. If lpName is NULL, the event
.text:77E344E0                                         ; object is created without a name. If lpName matches the name of
.text:77E344E0                                         ; another kind of object in the same name space (such as an existing
.text:77E344E0                                         ; semaphore, mutex, waitable timer, job, or file-mapping object), the
.text:77E344E0                                         ; function fails and the GetLastError function returns
.text:77E344E0                                         ; ERROR_INVALID_HANDLE. This occurs because these objects share the same
.text:77E344E0                                         ; name space.The name can have a "Global\" or "Local\" prefix to
.text:77E344E0                                         ; explicitly create the object in the global or session name space. The
.text:77E344E0                                         ; remainder of the name can contai
.text:77E344E3                 push    [ebp+lpEventAttributes] ;  A pointer to a SECURITY_ATTRIBUTES structure. If this parameter is
.text:77E344E3                                         ; NULL, the handle cannot be inherited by child processes. The
.text:77E344E3                                         ; lpSecurityDescriptor member of the structure specifies a security
.text:77E344E3                                         ; descriptor for the new event. If lpEventAttributes is NULL, the event
.text:77E344E3                                         ; gets a default security descriptor. The ACLs in the default security
.text:77E344E3                                         ; descriptor for an event come from the primary or impersonation token
.text:77E344E3                                         ; of the creator.
.text:77E344E6                 call    _CreateEventExA@16 ; Creates or opens a named or unnamed event object and returns a handle
.text:77E344E6                                         ; to the object.
.text:77E344EB                 pop     ebp
.text:77E344EC                 retn    10h
```

Figure 7.15: MSDN Comments Imported into IDA Pro.

When asked about the use of the color-coded stars, he did mention that numbers might be more useful and that highlighting the most recent would be helpful, with perhaps an exclamation mark. He also thinks that people will adapt to whichever colors or methods are used.

Interview 3: Tracks as an Authoring Tool

Mark finds the comments view pretty interesting and believes it would be useful with even only two to three people in a team. However, he would need to have a local server to support the comment threads since the machines used for analysis are standalone (not connected to the Internet) due to security reasons. He would also need to be able to link to the comments in IDA Pro, otherwise there would be two sets of comments spread out over two tools.

He does not think that the colors would be useful as they probably would not reach 5, 10 or 25 comments within their team. Mark mentions that a metric that might be better is the level of certainty of a comment. For example, red might be asking a question while green is a response to a question.

As for the granularity of commenting artifacts, he believes that for sequence diagrams, commenting calls, lifelines and cycles are pretty good. However, when he makes comments in IDA Pro, he often comments particular instructions.

Private documentation was seen as useful and especially so if it can be saved locally and shared offline.

An interesting use of the Tracks tool that Mark mentions is the ability to add and remove calls from both the static diagram and the dynamic traces. In this way, Tracks can be used as a sequence diagram authoring tool. Currently they are using Rational Rose [122], but he is not satisfied with it, and would also like to have integration with IDA Pro.

Interview Overview and Analysis

Table 7.5 shows a summary of the results from the interviews. We split up the results by question, but major points include the ability to have comments stored on a local server, being able to link comment threads with one another and with existing IDA Pro comments, being able to manually edit the sequence diagrams, and flexibility of what elements can have comments added to them (i.e. basic blocks).

Is there anything particularly useful in the demo?
History of comments
Post date to sort by newest comment
Ability to ask questions (have a discussion)
Useful even with a small team
Is there anything not useful?
Comments on mouseover
Would need a local server for standalone machines
Are there any features missing?
Navigate from comment to code
Automatically extract code/comments to report
Differences in comments between two executables
Link comment threads
Comment approval
Link with existing IDA Pro comments
Manually edit sequence diagram
Do you think the stars/colors are the best representation?
Number of comments
Highlighting most recent
Certainty of comment
Are there other artifacts you would like to be able to add comments to?
Basic blocks
Instructions
Do you think there is a need for private documentation in the same fashion?
100% (3 out of 3) Yes
Good place for rough notes
Save locally and share offline
How many comments do you foresee being in a thread?
Tree hierarchy or nested overlay

Table 7.5: Summary of Collaboration and Documentation Interviews.

7.4 Phase II and Phase III Limitations and Threats to Validity

As with any research study performed in an industrial setting, we faced challenges such as time constraints and intellectual property. We also may have threats to validity due to the research method used, as previously outlined in Section 4.1. This section further discusses these limitations as well as the external and internal threats to validity as they apply to both Phase II and III of this dissertation.

7.4.1 Limitations

The largest limitation was access to participants. In the mainframe group's case, we had to be very clear about the amount of time required for requirements elicitation. For example, the mainframe group's manager mistook the elicitation exercise as an all day exercise, rather than one to write notes down during the day: *"...for people onsite who will attend the meeting, are you expecting them to complete the handout? To be honest, I am not sure I can afford them the time to complete this beforehand"*. This misunderstanding was resolved with a follow-up phone call but it highlights the strict timelines with which we were bound.

In regard to elicitation within the malware group, while the team consisted of eight members, we were provided contact information for only six since they performed reverse engineering and were therefore relevant to our study. Of those six, we had four that responded to the user survey and subsequent elicitation exercise. The comment given during the group session was *"Some of my colleagues didn't really feel like entering because they thought it [the survey] was too vague in terms of questions. They would like more precise questions in the user survey, in order to get them back in"*. The process was completely voluntary in comparison to the mainframe group, the comment from the malware group's manager being that *"Their boss can tell them you go there and you do it, I can't really do that here"*.

We note that while our exploratory survey had 25 and 15 respondents for mainframe and malware respectively, our requirements elicitation had only 6 and 4. Previous work in usability studies found that statistically significant results are unlikely in a group size of less than eight participants, although it is possible [123]. Spyridakis and Fisher [124] found that between 10 and 12 participants will often produce statistical significance. Our smaller participant numbers are expected due to performing our study in industry, rather than with students, and each approach presents issues. While using groups of students would produce greater statistical significance with greater participant numbers, the data obtained would not be from target users. One possible technique used to produce statistical significance is *bootstrapping*. Bootstrapping creates new data samples by repeatedly choosing random values from the original sample set [125]. While this approach may be useful for modeling specific types of data, it would not have been useful in our particular study, which aimed to discover detailed requirement information from real users.

Another limitation we faced, was with the nature of the malware group as part of

a government defence facility. Due to security issues, we could not freely interview the team members about the nature of their daily work.

Finally, English was not the malware group's first language so there were times when discussion broke off in French (their primary language). At these points we would ask what the discussion was about, however it is possible we may have missed some important points. It is also possible that requirement descriptions were not explained as fully as they could have been.

7.4.2 External Validity

While each group was a practicing software group, they may be atypical of other groups. For example, the mainframe and malware groups each consisted of highly experienced developers working in a specific area. A different software group with less experience, or in a more generalized software field, could produce different results.

Additionally, our focus was on two assembly languages: HLASM and x86. The mainframe group, while not having an explicit notion of subroutines, was able to produce control flow data and therefore make the value for HLASM and Tracks known. Other combinations of groups may be able to use these seemingly isolated tools through similar means. While our results showed early on the issues with trying to consolidate differences between two languages, should our focus have been on x86 and perhaps ARM, the results may have appeared different.

7.4.3 Internal Validity

In regard to our requirements elicitation, the biggest threat to validity is possible confirmation bias, meaning that negative participant cues were ignored. We attempted to mitigate this by using double-blind responses and a coding scheme to determine the level of positivity in the results. In addition, the nominal session was scripted to ensure that we strictly adhered to the protocol. Finally, the nature of the study was not revealed to the participants, therefore changes in their behavior caused by an anticipation of what the experimenters were looking for should be minimized.

Another internal threat to validity with our requirements elicitation, is that participants were videotaped and aware of the researchers at all times, which may have introduced a *Hawthorne effect* [126]. This is when participants improve or modify an aspect of their behavior simply because they know they are being studied. This also

applied to the malware group, where certain information had to be kept secure at all times so participants may have needed to monitor their discussion.

Internal validity is also an issue since we were only able to spend a small amount of time with the groups. This affected our ability to isolate whether or not our requirements elicitation approach gave more requirements, or if the participants were naturally communicative.

In regard to the latter two internal validity threats, we attempted to isolate the effects of our approach by minimizing any additional factors that would alter motivation—participants were not directly involved with success of the process nor did they receive any judgemental feedback [126]. Participants also indicated they were used to informal meetings of this nature.

7.5 The Great Language Divide: Nature or Nurture?

We previously discussed two proof of concept tools that, unlike Tracks, could not be designed to meet requirements of both groups. Though it would appear that this stems solely from the groups' differences in terms of the daily tasks they perform, we believe our results support the argument that there are two factors. The first we see exemplified with LegaSee and is the existence of fundamental construct differences between these two particular assembly languages, and the nature of the instruction sets involved. The second is the specific daily tasks nurtured within the subcultures of the groups, compacted with the specific development/analysis environment.

The latter of the two is easily seen throughout our surveys, observations and requirements elicitation, presented as Phase II of this dissertation. In fact, Chapter 5 deals specifically with these differences in-depth. However, up until this point in the dissertation, we have not investigated how these two particular dialects—HLASM and x86—might have factored into the fracturing of the AVA framework.

We began this project with the simple research question: *Can program comprehension tools for high-level languages be retrofitted to apply to low-level languages?* This dissertation has shown that retrofitting high-level comprehension tools to low-level codebases can be a means of addressing a wide variety of requirements derived directly from developers working in this domain. However, by virtue of working with two stakeholders with a diverse set of needs, we were also able to explore a deeper

x86

```
.data

HelloWorldString:
  .ascii ''Hello World''

.text

.globl _start

_start:
  # Load all the
  # arguments for write ()

  movl $4, %eax
  movl $1, %ebx
  movl $HelloWorldString, %ecx
  movl $12, %edx
  int $0x80

  # Need to exit the program

  movl $1, %eax
  movl $0, %ebx
  int $0x80
```

ARM

```
.data

HelloWorldString:
  .ascii ''Hello World''

.text

.globl _start

_start:
  # Load all the
  # arguments for write ()

  mov r7, #4
  mov r0, #1
  ldr r1,=HelloWorldString
  mov r2, #12
  svc #0

  # Need to exit the program

  mov r7, #1
  mov r0, #0
  svc #0
```

HLASM [127]

```
EX1 CSECT
EX1 AMODE 31
EX1 RMODE 24

STM 14,12,12(13)
BALR 12,0
USING *,12

LA 5,WTO_AR
WTO TEXT=(5)
LMRET LM 14,12,12(13)

BR 14

IN_STRING DC C'Hello World'
WTO_AR DC AL2(L'OUT_STRING)
OUT_STRING DS CL(L'IN_STRING)
LTORG ,
END
```

Java

```
class HelloWorld {
  public static void
    main(String[] args) {
      System.out.println
        (''Hello World'');
  }
}
```

C

```
#include <stdio.h>

int main(void) {
  printf(''Hello World'');
  return 0;
}
```

Python

```
class HelloWorld():
  def __init__(self):
    print 'Hello World'
```

Table 7.6: "Hello World" Programs in Low- and High-Level Languages.

question, *Are tools in our proof of concept framework effective at supporting the requirements of both groups?* Though future research in intermediate representations to capture meaningful artifacts from a spectrum of languages may hold promise, our work has revealed that the fundamental disparity extends beyond the language constructs and into the nature of the work involved. In the case of our stakeholders, our results indicate that there would be no benefit in coalescing such disparate tools into one framework.

At the start of this project, we accepted the definition that assembly language could include any and all of the available dialects labelled as such. However, issues existed from the very beginning in even utilizing the same intermediate language for them due to issues surrounding instruction sets. We have come to the understand-

Characteristic	**x86**	**ARM**	**HLASM**
Instruction Set	CISC	RISC	IBM s370 CISC with Z-Series Additions
Opcodes	303	34	1505 [128]
General Registers	6 (16 bit)	16 (32 bit)	16 (64 bit)
Target Hardware	PC, Embedded	PC, Embedded	z/Architecture Mainframe
Subroutines	Yes	Yes	No
Structure		Areas	Modules, Sections, Classes, Elements, Parts
Macros	Yes	Yes	Yes
Unique Constructs			DSECTs, CSECTs

Table 7.7: Key Characteristics of x86, ARM and HLASM.

ing that the spectrum of dialects available under the umbrella "assembly language" taxonomy is potentially far more varied than even the spectrum for what we refer to as "high-level" languages. We would argue that the two languages we investigate in this dissertation are in fact on opposite ends of the spectrum. If we look at ARM in comparison to x86, we already begin to see more similarity than we do with HLASM, which could have produced very different results.

To further illustrate our point, we have provided Table 7.6 which shows the same "Hello World" program written in six different languages. Three of these languages are assembly: x86, ARM and HLASM; whereas three are high-level languages: Java, C and Python. We infer that even by looking at a simple program, we can immediately see more similarity between x86 and ARM, as well as between all three high-level languages, than we can between x86 and HLASM. In fact, HLASM is a major outlier.

While these similarities may only be syntactic, we provide yet another comparison in Table 7.7. This table shows different factors which could be used in comparing programming languages. We observe that there exist marked differences between all three assembly languages, which include instruction sets, in addition to lack of fundamental constructs such as subroutines. We originally labelled x86 and HLASM as different assembly language dialects, but have since come to the realization that they were, in fact, different languages altogether.

7.6 Chapter Summary

This chapter began by providing technical challenges and limitations. We then showed two tools that are part of the AVA framework, LegaSee and REwind, that have only shown promise to one of the two groups, mainframe and malware respectively. These two tools exemplify how specific the two domains really are. While these tools do not show promise universally, that does not preclude the existence of tools that do apply to both groups. Tracks solves problems that exist in both domains by providing additional support for the mainframe by superimposing functional decomposition on the codebase. This chapter provided limitations and threats to validity of our study. Phase III was concluded by answering our final research question: *Are tools in our proof of concept framework effective at supporting the requirements of both groups?*

Table 7.8 shows a summary of each tool split into each category defined from the requirements elicitation which took place in Chapter 4. This table highlights the fact that while a category at the high-level may seem like the same problem, when we dig deeper, we see that the issues that exist within that category for each group are quite unique.

Tool	Requirement Category	Mainframe	Malware
Tracks	Browsing and Navigation	Navigation to Listing	Navigation within IDA Pro, Navigation History View
	Control Flow	Static Control Flow, Reversed Static, Trace Log	Static Control Flow, Reversed Static
	Data		(Data Required to Reach Execution Points)
	Debugging	(Multi-Threading Support)	Dynamic Control Flow, (Compare Traces)
	De-obfuscation		Cycle/Loop Detection, API Call Patterns
	Documentation	Module Descriptions, Comment Threads, Save to Image File, (Error Code Reference)	MSDN Documentation, Comment Threads, Save to Image File,
	Integration	(Socket Message Capable)	IDA Pro, (Filter with PaiMei)
	References		DLLs Referenced
LegaSee	Browsing and Navigation	High-Level System View, Navigation to DSECTs and CSECTs	
	Data	DSECT Module Location and Size	
	References	CSECT Module Location and Size	
REwind	Data		(Re-running with New Data)
	Debugging		Repeat Debugging Actions
	De-obfuscation		Discover Malware Intent through Iterative Execution, Anti-Debugging Trap Avoidance
	Documentation		State Diagram, State Descriptions
	Integration		IDA Pro

Table 7.8: Summary of Group Requirements Supported by Tool[a].

[a]Items in brackets are yet to be implemented. Categories without points are not shown.

Chapter 8

Future Research Directions and Conclusions

This final chapter concludes the main claim of this dissertation, *while program comprehension tools can be effectively applied to low-level programming languages, such as assembly language, they cannot be universally applied due to their specialized use in industrial software groups, compounded by fundamental construct differences.* This chapter discusses future research directions stemming from this work, including further data analysis from our requirements elicitation, as well as further tool development towards these requirements. We conclude by revisiting the research questions and results of this dissertation.

8.1 Future Research Directions

There are many avenues of future research that could result from the work provided in this dissertation. We cover these avenues by their topics: requirements elicitation, feature additions to existing tools, integration with other assembly language tool support or work environments, and user studies that could take place to validate such tools.

8.1.1 Analysis of Requirements Elicitation Data

In terms of future research for disseminating the data collected as part of our requirements elicitation process, much more could be done. For example, we have numerical data for scores, and have shown graphs for the preliminary and final rankings and

have given a possible reasoning for fluctuations of scores. However, given this data, evaluations of how scores changed over time could be done.

While we use Interaction Process Analysis (IPA) on the transcribed audio from our group sessions, we could perform additional qualitative data analysis by using the transcribed audio with software which supports grouping and analyzing textual data such as ATLAS.ti [129]. We could then compare these results with that of our study to see if any requirements were not explicitly reported. For example if the words "seg fault" had been mentioned noticeably often, we could deduce that they are an area of interest despite perhaps not being mentioned as a specific issue.

8.1.2 Tools to Satisfy Elicited Requirements

This dissertation has brought to light many issues that assembly language developers and analysts face. We have only implemented proof of concept tools to satisfy a small portion of these requirements[1]. However, there are additional features for the tools we have already implemented that could be added to satisfy even more of these requirements. We discuss those feature additions below, in terms of each tool they could be integrated into.

Tracks

Our exploratory surveys, as well as requirements elicitation and interviews have identified multiple features that could be added to the Tracks tool. These include:

1. recognizing (suspicious) system call patterns
2. documentation support (adding notes, modifying diagram for reports)
3. comparing traces
4. multi-threading support
5. data required to reach execution points
6. calculating branch frequency
7. performance statistics

The system call patterns can be a good indicator as to whether or not a program is malicious. For example, we could mark a cycle as *Decryption*. In reality, these call patterns would likely be obfuscated and being able to automatically detect whether

[1]These requirements are examined in-depth in Chapters 3 and 4 so we do not revisit them here.

or not code is malicious would be non-trivial. Therefore, the malicious call pattern would still need to be recognized by a human and pattern data could be grown over time, or this functionality could be used as a training tool. As for documentation support, like Code Bubbles, we feel that it is important, when faced with so much information, to be able to make notes and flag items from within the tool. This is something we would like to contend with both in Tracks and future tools. Comparing two traces to see how the program executes differently from one run to the next is very important as well. It can show which data to use to execute a particular scenario. Lastly, branch frequency would indicate how often specific code is run. This can be helpful to locate performance bottlenecks.

Comment Threads

There are many avenues of future work for the comment capabilities within Tracks. While we discuss these in terms of how they connect with Tracks' sequence diagrams, these features are also important for comment integration across all of AVA's suite of tools.

Since most users currently create their comments inline in the assembly code within IDA Pro, we would need to be able to export those into the comment threads. Another important feature will be the use of sequence diagrams within reports. For example, the ability to link a code snippet to the comment thread so that it can be exported to documentation may be of importance.

As for the granularity of what can be commented, we need to be able to support some level of arbitrary comment placement. For example, users need to comment parts of the code such as basic blocks and particular instructions. We need to indicate placement of such comments within the sequence diagram. To alert the users to the presence of comments, we may look at metrics other than the number. Other metrics that presented themselves were the most recent posts as well as the certainty of the post (e.g. question versus answer).

We will also need to consider the scalability of comment threads and the amount by which the threads will grow in practice. If size becomes an issue, we may need to look at other ways to represent them, such as with a tree hierarchy or with nested overlays. We may also need to have the comments window show up on top of the sequence diagram if we do not want to interrupt the work flow in a large trace. As for the comment capability itself, the ability to categorize, tag and search for posts will

need to be added. The tagging can lend itself to inclusion within other documents but also to link comment threads and be used for searching. We need to allow users to link to external information such as internet links, and files, as well as provide support for private documentation that can be saved locally and shared offline. This gives the user a private area in which to save notes. This might be implemented as a separate tab within the comments view in much the same way as the collaborative approach.

Other areas of future work include updates on what other analysts are doing, support for comments on a private server, retracing steps that an analyst took, the difference listing between comments on two executables and comment approval.

References to Documentation

We provided integration of the MSDN library to Tracks at the request of malware analysts. Therefore the documentation that was linked to Tracks' diagrams and external functions was the MSDN documentation. There are two avenues of future work for these features. The first is taking the data offline, and the second is integrating further documentation sources.

The first point is integral to the nature of the work performed by the malware analysts. Since they reverse engineer malware, it is imperative that these analysis machines stay disconnected from any network, most definitely including the Internet. This made our current approach of using Google to search the MSDN website of little use. While our approach was to show how this feature could potentially work, and to gauge its usefulness, this functionality will need to be updated to use locally saved MSDN documentation. There are two possibilities we see for this: saving the entire website and indexing it using Google Desktop, or somehow accessing the MSDN information provided as part of the Visual Studio installation.

As far as integrating further documentation sources, this should ideally be done in a way that is independent of the documentation. If we use the desktop-indexing method as discussed in the previous paragraph, we could provide results across multiple documentation resources. Further, we can incorporate documentation independent of the system the diagram was built for. For example, the mainframe developers often need to look up mainframe error codes. If we were able to capture this information within our data sources, regardless of whether or not this was shown in the diagram, we could then load pertaining documentation based on search.

8.1.3 Integration of AVA with Other Systems

While our AVA framework is implemented in such a way as to make it independent of the tools it interfaces with, we have only integrated it directly with IDA Pro at this stage. Since IDA Pro was the primary tool used by the malware analysts, it was already a preferred option. However, what allowed us to create the integration easily was the plugin capability it provides. While this is not a prerequisite for working with our framework, since we only require messages to be sent over sockets, it created an avenue for us to quickly realize a result. We had originally planned to show a similar integration with the mainframe environment, however that would have required access to their systems, as well as a thorough understanding of it. Initial plans were to have students involved in this domain to implement this messaging service, however when our CA research contact left, we also lost access to these resources. We therefore know that such integration is a possibility, and would be a further proof of applicability of our tools within other domains.

8.1.4 User Studies of Proof of Concept Tools

We provide an assessment of AVA in Chapter 7, which shows case studies of how the framework's tools can be applied within the two industrial domains we investigate. In this assessment, we show how the Tracks tool was able to solve issues present within both groups, whereas both the LegaSee and REwind tools could not. The claim of this dissertation is to explore whether or not high-level comprehension tools could be applied to two specific low-level assembly languages. We therefore required the implementation of these tools to illustrate our claim.

However, we are aware that the Tracks tool is already being used in industry, and we have received preliminary feedback from its use. We recognize that user studies are necessary to assess how these tools need to evolve, as well as to compare how they perform in comparison to existing tools. The results of such a user study would be one avenue of future work to further validate our requirements elicitation process.

8.2 Conclusions

In this dissertation, we were able to demonstrate why understanding certain aspects of low-level languages, such as assembly language, can be far more challenging than high-level languages. For example, control flow contains unstructured branching, and in

some cases, a lack of functions altogether. While we know these issues exist, currently there is little in the way of tool support to aid with these program comprehension issues.

We began this journey asking whether or not high-level program comprehension tools could be effectively applied to low-level languages. To do so, we had to elicit requirements from two specialized groups: a mainframe development group using HLASM, and a government agency that analyzes malware threats, predominantly in x86. This in turn led us to a deeper investigation. Our original claim was that, *while program comprehension tools can be effectively applied to low-level programming languages, such as assembly language, they cannot be universally applied due to their specialized use in industrial software groups, compounded by fundamental construct differences.*

In the process of demonstrating our claim, we showed the following:

Phase II Surveys and requirements elicitation, presented in Chapters 3 and 4 respectively, showed there exists a minimal intersection of requirements between these two highly-specialized industrial software groups. We compared the results of each side-by-side in Chapter 5, and reason about why the differences exist. This phase answered our first two research questions: *What are the requirements currently not being met in the comprehension of assembly code within two unique groups: mainframe developers and malware analysts?* and *What are the similarities and differences in the requirements?*

We started by issuing an exploratory survey to better understand the wide range of issues involving assembly language comprehension within each of these groups. We next conducted requirements elicitation studies within two industrial groups using techniques adopted from social psychology. We showed that these techniques from social psychology can be used successfully in a highly-specialized industrial development context, without the techniques being deemed as intrusive or irrelevant. Further, we provided a ranked list of requirements and descriptions, in order to build proof of concept tools. While both the survey and requirements elicitation provided rich details for each group, we noticed that, while issues each group faced seemed similar at a high-level, when we dug deeper, the specific issues they faced are in fact quite different. We dedicated all of Chapter 5 to discussing the similarities and differences between each group, giving an explanation as to why these exist. These differences are key in how the program comprehension framework we call AVA was built.

Phase III, Part I In Chapter 6, we provided the design and implementation for tools within the AVA framework: LegaSee, REwind, and Tracks. These tools answered our third research question: *Can program comprehension tools for high-level languages be retrofitted to apply to low-level languages?*

We discussed in-depth how AVA was created with independence of the underlying assembly language in mind. However, even with this being the case, certain tools can only meaningfully exist in a space where there is a legitimate problem for them to solve. While tools can be built that can solve problems universally, there are others which cannot.

Phase III, Part II Chapter 7 illustrated case studies using the three tools in the AVA framework. Of these three tools, only the Tracks tool has been successful at addressing issues universally. Tracks is a control flow tool we have built that can show control flow sequence diagrams for both HLASM and x86 code, and has been built with mainframe and malware issues in mind. We show in our assessment how it can be used for both groups, and also showcase some of the features helpful for each. While Tracks cannot fix inherent issues with irreducible control flow or "spaghetti-code", it does reduce the cognitive overhead for large traces, as well as provides assistance with both navigation and de-obfuscating API calls.

We additionally showed two tools that could not be applied to both groups, LegaSee and REwind. LegaSee shows a high-level construct view for DSECTs and CSECTs within HLASM, and REwind provides the ability to repeat debugging steps in the course of reverse engineering malware threats. The reasons for this lack of universal applicability are two-fold. First, the work conducted by each group differs significantly, meaning that issues simply do not exist for both groups. Second, the languages themselves differ along so many facets that they are very distinct from one another. These case studies answered our final research question: *Are tools in our proof of concept framework effective at supporting the requirements of both groups?*

As with any research study, we may have limitations and threats to validity due to the research method used. Our largest limitation was access to participants, as well as with intellectual property issues surrounding the nature of the stakeholder groups. Additionally, our study only took into account two highly-specialized groups which each used their own assembly language: HLASM or x86. It is possible that with different groups, or with different assembly languages, our results would have appeared different. For example, ARM was investigated as a possible intermediate language

candidate that showed some promise for x86, however the inclusion of HLASM made this an impossibility [130]. Should our case studies have been ARM and x86, we may have been able to find more common ground, though ultimately the truth remains the same—there exists no silver bullet to reconcile issues across all assembly language domains.

Ultimately, our AVA (Assembly Visualization and Analysis) program comprehension framework comprises tools that can work independently of the underlying language by using intermediate data formats, as well as communication mechanisms to interface with external systems. While all of the tools we build aim to be flexible and language agnostic, the bottom line is that there are fundamental disparities between the two groups that make a universal approach unrealistic. There are two main reasons for these disparities. First, the assembly languages used are technically quite different from one another and therefore contain fundamental construct differences between them. Second, these highly-specialized groups perform specific work tasks, within specific development and analysis environments. Despite these disparities, we have shown that tools do exist that can work for both groups in our study. Additionally, we provided requirements that can spawn further tool development and analysis. Through AVA, we have provided the ability for future researchers to expand upon our work by integrating the framework with not only their own assembly languages, but also their specific systems.

Bibliography

[1] L. M. Surhone, M. T. Tennoe, and S. F. Henssonow, *IBM High Level Assembler (HLASM)*. Mauritius: Betascript Publishing, 2010.

[2] D. B. Kahanwal, "Abstraction level taxonomy of programming language frameworks," *International Journal of Programming Languages and Applications (IJPLA)*, vol. 3, no. 4, 2013.

[3] IBM Corporation, *HLASM: V1R6 Language Reference*, 1982-2000.

[4] R. Marik, "GREX Architecture - Package Comprehension Report," tech. rep., CA Labs, July 2009.

[5] C. Eagle, *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler*. San Francisco, CA, USA: No Starch Press, 2008.

[6] P. Sinha, A. Boukhtouta, V. H. Belarde, and M. Debbabi, "Insights from the Analysis of the Mariposa Botnet," in *5th International Conference on Risks and Security of Internet and Systems (CRISIS)*, (Montreal, Quebec, Canada), 2010.

[7] C. Stangor, *Social groups in action and interaction*. Psychology Press, 2004.

[8] I. L. Janis, *Groupthink: psychological studies of policy decisions and fiascoes*. Boston: Houghton Mifflin, 1982.

[9] H. Tajfel, M. G. Billig, R. P. Bundy, and C. Flament, "Social categorization and intergroup behaviour," *European Journal of Social Psychology*, vol. 1, no. 2, pp. 149–178, 1971.

[10] T. Postmes, R. Spears, and S. Cihangir, "Quality of decision making and group norms.," *Journal of Personality and Social Psychology*, vol. 80, no. 6, pp. 918–930, 2001.

[11] A. Teh, E. Baniassad, D. van Rooy, and C. Boughton, "Social psychology and software teams: A preliminary look at establishing task-effective group norms," *IEEE Software*, vol. 99, no. PrePrints, 2011.

[12] A. W. Kruglanski, "Motivations for judging and knowing: Implications for causal attribution," *The handbook of motivation and cognition: Foundation of social behavior*, vol. 2, pp. 333 – 368, 1990.

[13] C. H. Hui, "Measurement of individualism-collectivism," *Journal of Research in Personality*, vol. 22, no. 1, pp. 17 – 36, 1988.

[14] A. W. Kruglanski and D. M. Webster, "Motivated closing of the mind:"seizing" and "freezing"," *Psychological Review*, vol. 103, no. 2, pp. 263–283, 1996.

[15] M. N. Bechtoldt, C. K. W. De Dreu, B. A. Nijstad, and H.-S. Choi, "Motivated information processing, social tuning, and group creativity," *Journal of Personality and Social Psychology*, vol. 99, no. 4, pp. 622–637, 2010.

[16] D. Oyserman, H. M. Coon, and M. Kemmelmeier, "Rethinking individualism and collectivism: evaluation of theoretical assumptions and meta-analyses," *Psychological Bulletin*, vol. 128, no. 1, pp. 3–72, 2002.

[17] J. Goncalo and B. Staw, "Individualism-collectivism and group creativity," *Organizational Behavior and Human Decision Processes*, vol. 100, no. 1, pp. 96–109, 2006.

[18] R. Griffin, *Management*. Houghton Mifflin Co., 2006.

[19] C. Okoli, "The delphi method as a research tool: an example, design considerations and applications," *Information & Management*, vol. 42, no. 1, pp. 15–29, 2004.

[20] "Architecture-Driven Modernization (ADM): Knowledge Discovery Meta-Model (KDM)," Object Management Group, 2009.

[21] P. Amini, "PaiMei - Reverse Engineering Framework," in *RECON '06: Reverse Engineering Conference*, (Montreal, Canada), 2006.

[22] HBGary, "Responder Pro." `https://www.hbgary.com/products-services/responder-pro`, 2010.

[23] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "Bitblaze: A new approach to computer security via binary analysis," in *ICISS '08: Proceedings of the 4th International Conference on Information Systems Security*, (Hyderabad, India), pp. 1–25, Springer-Verlag, 2008.

[24] "Zynamics." `http://www.zynamics.com`, 2010.

[25] "ASMPlugin." `http://sourceforge.net/projects/asmplugin`, 2010.

[26] P. Borunda, C. Brewer, and C. Erten, "GSPIM: graphical visualization tool for MIPS assembly programming and simulation," in *SIGCSE '06: Proceedings of the 37th SIGCSE technical symposium on Computer science education*, (New York, NY, USA), pp. 244–248, ACM, 2006.

[27] R. N. Horspool, W. D. Lyons, and M. Serra, "ARMSim# - a Customizable Simulator for Exploring the ARM Architecture," in *FECS '09: Proceedings of the 2009 World Congress in Computer Science, Computer Engineering and Applied Computing*, (Las Vegas, NV, USA), July 2009.

[28] "TextMaestro." `http://www.textmaestro.com`, 2010.

[29] "The collabREate Project." `http://www.idabook.com/collabreate/`, 2010.

[30] C. Eagle and T. Vidas, "Next Generation Collaborative Reversing with Ida Pro and CollabREate," in *Black Hat Briefings*, (Las Vegas, USA), August 2008.

[31] B. Cleary, M.-A. Storey, L. Chan, M. Salois, and F. Painchaud, "Atlantis - assembly trace analysis environment," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pp. 505–506, 2012.

[32] J. Bohnet and J. Döllner, "Visual exploration of function call graphs for feature location in complex software systems," in *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization*, (New York, NY, USA), pp. 95–104, ACM, 2006.

[33] N. Synytskyy, R. C. Holt, and I. Davis, "Browsing software architectures with LSEdit," in *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, (Washington, DC, USA), pp. 176–178, IEEE Computer Society, 2005.

[34] E. R. Gansner and S. C. North, "An open graph visualization system and its applications to software engineering," *Softw. Pract. Exper.*, vol. 30, no. 11, pp. 1203–1233, 2000.

[35] T. Munzer, *Interactive Visualization of Large Graphs and Networks*. Phd dissertation, Stanford University, Palo Alto, California, 2000.

[36] "aiSee Graph Layout Software." `http://www.aisee.com`, 2010.

[37] S. Ducasse, T. Girba, and A. Kuhn, "Distribution map," in *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, (Washington, DC, USA), pp. 203–212, IEEE Computer Society, 2006.

[38] R. DeLine, "Staying Oriented with Software Terrain Maps," in *Workshop on Visual Languages and Computation*, pp. 309–314, 2005.

[39] M.-A. Storey, K. Wong, and H. A. Müller, "Rigi: a visualization environment for reverse engineering," in *ICSE '97: Proceedings of the 19th international conference on Software engineering*, (Boston, Massachusetts, United States), pp. 606–607, ACM, 1997.

[40] A. Desnos, S. Roy, and J. Vanegue, "ERESI: a kernel-level binary analysis framework," in *SSTIC '08: Symposium sur la Securite des Technologies de l'Information et Communications*, (Rennes, France), 2008.

[41] Q. Wang, W. Wang, R. Brown, K. Driesen, B. Dufour, L. Hendren, and C. Verbrugge, "EVolve: an open extensible software visualization framework," in *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, (New York, NY, USA), pp. 37–ff, ACM, 2003.

[42] M.-A. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. Musen, "SHriMP views: an interactive environment for information visualization and navigation," in *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, (New York, NY, USA), pp. 520–521, ACM, 2002.

[43] M. Eichberg, M. Haupt, and M. Mezini, "The SEXTANT Software Exploration Tool," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 753–768, 2006.

[44] J. van Wijk and H. van de Wetering, "Cushion treemaps: visualization of hierarchical information," in *Information Visualization, 1999. (Info Vis '99) Proceedings. 1999 IEEE Symposium on*, pp. 73–78, 147, 1999.

[45] "Disk Inventory X." `http://www.derlien.com`, 2010.

[46] S. Boccuzzo and H. C. Gall, "CocoViz: Supported Cognitive Software Visualization," in *WCRE '07: Proceedings of the 14th Working Conference on Reverse Engineering*, (Washington, DC, USA), pp. 273–274, IEEE Computer Society, 2007.

[47] R. Wettel and M. Lanza, "CodeCity: 3D visualization of large-scale software," in *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, (New York, NY, USA), pp. 921–922, ACM, 2008.

[48] G. de F. Carneiro, R. Magnavita, E. Spinola, F. Spinola, and M. Mendonça, "An Eclipse-Based Visualization Tool for Software Comprehension," in *Tools Session of the Brazilian Symposium on Software Engineering (SBES'2008)*.

[49] J. Malnati, "X-Ray: An Eclipse Plug-in for Software Visualization," Master's thesis, Lugano University, 2007.

[50] K. De Volder, "JQuery: A generic code browser with a declarative configuration language," in *PADL 2006: Proceedings of the 8th International Symposium on Practical Aspects of Declarative Languages*, (Charleston, SC), pp. 88–102, 2006.

[51] D. Poshyvanyk, A. Marcus, and Y. Dong, "JIRiSS - an Eclipse plug-in for Source Code Exploration," in *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension*, (Washington, DC, USA), pp. 252–255, IEEE Computer Society, 2006.

[52] M. Marin, L. Moonen, and A. van Deursen, "FINT: Tool Support for Aspect Mining," in *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering*, (Washington, DC, USA), pp. 299–300, IEEE Computer Society, 2006.

[53] J. Hannemann and G. Kiczales, "Overcoming the Prevalent Decomposition in Mainframe Code," in *Workshop on Advanced Separation of Concerns, Int'l Conf. Software Engineering (ICSE)*, 2001.

[54] P. Tonella and M. Ceccato, "Aspect Mining through the Formal Concept Analysis of Execution Traces," in *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, (Washington, DC, USA), pp. 112–121, IEEE Computer Society, 2004.

[55] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," in *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, (Washington, DC, USA), p. 301, IEEE Computer Society, 2001.

[56] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 210–224, 2003.

[57] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Transactions on Software Engineering Methodology (TOSEM)*, vol. 16, no. 1, p. 3, 2007.

[58] D. A. Quist and L. M. Liebrock, "Visualizing compiled executables for malware analysis," in *The 6th International Symposium on Visualization for Cyber Security (VizSec)*, 2009.

[59] R. DeLine, M. Czerwinski, B. Meyers, G. Venolia, S. M. Drucker, and G. G. Robertson, "Code Thumbnails: Using Spatial Memory to Navigate Source Code," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, (Brighton, UK), pp. 11–18, IEEE Computing Society, 2006.

[60] D. Macdonald, "Zeus: God of DIY Botnets," *FortiGuard Center — Threat Research and Response*, Oct. 2009.

[61] H. Binsalleeh, T. Ormerod, A. Boukhtouta, S. Prosenjit, A. Youssef, M. Debbabi, and L. Wang, "On the Analysis of the Zeus Botnet Crimeware Toolkit," in *8th Annual Conference on Privacy, Security and Trust (PST)*, (Ottawa, Ontario, Canada), 2010.

[62] A. Bragdon, S. P. Reiss, R. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," in *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, (New York, NY, USA), pp. 455–464, ACM, 2010.

[63] C. Bennett, D. Myers, M.-A. Storey, and D. M. German, "Working with 'monster' traces: Building a scalable, usable, sequence viewer.," in *In Proceedings of the 3rd International Workshop on Program Comprehension Through Dynamic Analysis (PCODA)*, (Vancouver, Canada), pp. 1–5, 2007.

[64] M. McGavin, T. Wright, and S. Marshall, "Visualisations of execution traces (VET): an interactive plugin-based visualisation tool," in *AUIC '06: Proceedings of the 7th Australasian User interface conference*, (Hobart, Australia), pp. 153–160, 2006.

[65] "Sysersoft." `http://www.sysersoft.com`, 2010.

[66] J. Russell and R. Cohn, *Ollydbg*. Book on Demand, 2012.

[67] "SoftICE." `http://en.wikipedia.org/wiki/SoftICE`, 2010.

[68] "Corelabs site." `http://corelabs.coresecurity.com`, 2010.

[69] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO '04: Proceedings of the International Symposium on Code generation and Optimization*, (Washington, DC, USA), p. 75, IEEE Computer Society, 2004.

[70] T. Dullien and S. Porst, "REIL : A platform-independent intermediate representation of disassembled code for static code analysis," *In Proceeding of CanSecWest*, 2009.

[71] D. W. Pucsek, "Visualization and analysis of assembly code in an integrated comprehension environment," Master's thesis, University of Victoria, 2013.

[72] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot, "MoDisco: A generic and extensible framework for model driven reverse engineering," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, (New York, NY, USA), pp. 173–174, ACM, 2010.

[73] J. desRivieres and J. Wiegand, "Eclipse: A platform for integrating development tools," *IBM Systems Journal*, vol. 43, no. 2, pp. 371–383, 2004.

[74] C. Bennett, D. Myers, M.-A. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland, "A survey and evaluation of tool features for understanding

reverse-engineered sequence diagrams," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 20, no. 4, 2008.

[75] R. Harris and R. Warner, *The definitive guide to SWT and JFace*. Apress, 2004.

[76] C. Bennet, "Tool features for understanding large reverse engineered sequence diagrams," Master's thesis, University of Victoria, 2008.

[77] *Eclipse Development Using the Graphical Editing Framework and the Eclipse Modeling Framework*. Riverton, NJ, USA: IBM Corp., 2004.

[78] A. Clement, A. Colyer, and M. Kersten, "Aspect-oriented programming with AJDT," in *ECOOP Workshop on Analysis of Aspect-Oriented Software*, 2003.

[79] J. Baldwin and Y. Coady, "Adaptive Systems Require Adaptive Support - When Tools Attack!," in *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*, p. 10, 2007.

[80] R. Likert, "A Technique for the Measurement of Attitudes," *Archives of Psychology*, vol. 22, no. 140, p. 55, 1932.

[81] Norman Sandbox, "Norman Sandbox Whitepaper," tech. rep., 2003. `http://www.norman.com/documents/wp_sandbox.pdf`.

[82] Sunbelt Software, Inc., "CWSandbox." `http://www.sunbeltsoftware.com/Malware-Research-Analysis-Tools/Sunbelt-CWSandbox`, 2010.

[83] R. Brooks, "Towards a theory of the comprehension of computer programs.," in *International Journal of Man-Machine Studies*, vol. 18, pp. 534–554, 1983.

[84] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, (New York, NY, USA), pp. 1–11, ACM, 2006.

[85] K. Erdös and H. M. Sneed, "Partial comprehension of complex programs (enough to perform maintenance)," in *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, (Washington, DC, USA), p. 98, IEEE Computer Society, 1998.

[86] J. Baldwin, P. Sinha, M. Salois, and Y. Coady, "Progressive user interfaces for regressive analysis: Making tracks with large, low-level systems," in *Proceedings of the Australasian User Interface Conference (AUIC)*, (Perth, Australia), 2011.

[87] C. Treude, F. Figueira Filho, M.-A. Storey, and M. Salois, "An exploratory study of software reverse engineering in a security context," in *18th Working Conference on Reverse Engineering (WCRE)*, pp. 184 –188, Oct. 2011.

[88] J. Baldwin, A. Teh, E. Baniassad, D. van Rooy, and Y. Coady, "Requirements for comprehension tools to help highly-specialized industrial software groups and how to elicit these requirements," *In submission to Requirements Engineering*, 2014.

[89] "LimeSurvey: An Open Source survey tool." `http://www.limesurvey.org`, 2012.

[90] D. M. Webster and A. W. Kruglanski, "Individual differences in need for cognitive closure," *Journal of Personality and Social Psychology*, vol. 67, no. 6, pp. 1049–1062, 1994.

[91] A. Roets and A. van Hiel, "Item selection and validation of a brief, 15-item version of the need for closure scale," *Personality and Individual Differences*, vol. 50, no. 1, pp. 90–94, 2011.

[92] K. A. Ericsson and H. Simon, *Protocol analysis: verbal reports as data*. Cambridge, MA: MIT press, 1993.

[93] C. Lewis and J. Rieman, *Task-Centered User Interface Design:A Practical Introduction*. 1994.

[94] J. Goguen and C. Linde, "Techniques for requirements elicitation," in *Proceedings of IEEE International Symposium on Requirements Engineering (RE)*, pp. 152 –164, Jan. 1993.

[95] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *Proceedings of the Centre for Advanced Studies Conference (CASCON)*, IBM Press, 1997.

[96] A. L. Delbecq and A. H. van de Ven, "A Group Process Model for Problem Identification and Program Planning," *Journal Of Applied Behavioral Science VII*, pp. 466 –91, 1971.

[97] M. Diehl and W. Stroebe, "Productivity loss in brainstorming groups: Toward the solution of a riddle," *Journal of Personality and Social Psychology*, vol. 53, no. 3, pp. 497–509, 1987.

[98] M.-A. Storey, L.-T. Cheng, I. Bull, and P. Rigby, "Shared waypoints and social tagging to support collaboration in software development," in *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, (New York, NY, USA), pp. 195–198, ACM, 2006.

[99] C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Tech. Rep. 148, Department of Computer Sciences, The University of Auckland, New Zealand, July 1997.

[100] R. F. Bales, *Interaction Process Analysis*. 1950.

[101] A. Teh, *Normative Manipulation as a way of Improving the Performance of Software Engineering Groups: Three Experiments*. PhD thesis, The Australian National University, 2012.

[102] "First Nations' Stewardship Tools Partnership." `http://web.uvic.ca/fnst/`, 2013.

[103] S. Micallef, *IDA PLUG-IN WRITING IN C/C++*. 1.1 ed., 2009.

[104] "CBT Tape - MVS Freeware." `http://www.cbttape.org`, 2010.

[105] A. S. Tanenbaum, "A tutorial on algol 68," *Computing Surveys*, vol. 8, 1976.

[106] "Introducing ASMPUT." `http://pic.dhe.ibm.com/infocenter/zos/v1r11/index.jsp?topic=/com.ibm.zos.r11.asmk200/putintr.htm`, 2013.

[107] P. Charland, D. Dessureault, D. Ouellet, and M. Lizotte, "Opening up architectures of software-intensive systems: A first prototype implementation," Technical Memorandum TM 2006-781, DRDC Valcartier, 2007.

[108] "Plug-In Contest 2011: Hall Of Fame." `http://www.hex-rays.com/contests/2011/index.shtml`, 2012.

[109] J. Baldwin and Y. Coady, "Social security: collaborative documentation for malware analysis," in *Proceedings of the 12th Annual Conference of the New Zealand Chapter of the ACM Special Interest Group on Computer-Human Interaction*, CHINZ '11, (New York, NY, USA), pp. 17–24, ACM, 2011.

[110] J. Russell and R. Cohn, *Google Sidewiki*. Book on Demand, 2012.

[111] M. Thompson, "Mariposa botnet analysis," tech. rep., Defence Intelligence, 2010.

[112] D. Sanderson, *Programming Google App Engine*. 2009.

[113] J. Baldwin and Y. Coady, "AVA: Assembly Visualization and Analysis," in *Eclipse Demo Camp*, (Vancouver, BC, Canada), June 2012.

[114] J. Ryall, D. Myers, and J. Anvik, "TagSEA for IDA." `http://thechiselgroup.org/2012/07/19/tagsea-for-ida/`, 2006.

[115] M. van Emmerik and T. Waddington, "Using a decompiler for real-world source recovery," in *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, (Washington, DC, USA), pp. 27–36, IEEE Computer Society, 2004.

[116] "IDA Plugins: Sobek." `http://www.openrce.org/downloads/details/38/Sobek`, 2012.

[117] N. Falliere, "Windows anti-debug reference," *Symantec Connect*, September 2007.

[118] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware analysis via hardware virtualization extensions," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, (New York, NY, USA), pp. 51–62, ACM, 2008.

[119] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *J. Comput. Secur.*, vol. 6, no. 3, pp. 151–180, 1998.

[120] M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala, "Malware detection using assembly and API call sequences," *Journal in Computer Virology*, April 2010.

[121] S. Porst, "Importing MSDN documentation into IDA Pro," in *Official Zynamics Company Blog*, 2010. http://blog.zynamics.com/2010/04/30/importing-msdn-documentation-into-ida-pro/.

[122] T. Quatrani, *Visual modeling with rational Rose and UML.* Addison-Wesley object technology series, Addison-Wesley-Longman, 1998.

[123] J. Nielsen and T. K. Landauer, "A mathematical model of the finding of usability problems," in *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, (New York, NY, USA), pp. 206–213, ACM, 1993.

[124] J. H. Spyridakis and J. R. Fisher, "Usability testing in technical communication: The application of true experimental designs," in *Technical Communication*, pp. 469–72, 1992.

[125] D. Shasha and M. Wilson, *Statistics is Easy!* Synthesis lectures on mathematics and statistics, Morgan & Claypool Publishers, 2008.

[126] R. H. Franke and J. D. Kaul, "The hawthorne experiments: First statistical interpretation," *American Sociological Review*, vol. 43, no. 5, pp. pp. 623–643, 1978.

[127] "Introduction to HLASM," in *SHARE - High Level Assembler Bootcamp*, (Boston), IBM UK, 2013.

[128] "System z Instructions Mnemonic List." http://www.tachyonsoft.com/inst390m.htm, 2013.

[129] S. Friese, *Qualitative Data Analysis with ATLAS.ti.* SAGE Publications, 2011.

[130] J. Wall, "ARM Assembly Language as an Intermediate Representation for Multiple CISC Assembly Languages," work term report, University of Victoria, 2009.

Appendix A

Exploratory Survey Summary

This appendix provides a summary reference for the large exploratory survey in Chapter 3. Table A.1 summarizes the results of the survey for the mainframe group, while Table A.2 shows the results for the malware analysts. Each top-level row refers to each section of the survey.

Section	Topic	Result
About the Participant	Dev Experience	88% 10+ Years
	Most Familiar PL	100% Assembly, 48% REXX, 48% C/C++
	Favorite PL	56% Assembly
	Favorite Tool	36% Text editor, 32% Debugger
Assembly Experience	Writing	4.42/5
	Understanding	4.46/5
	Most Familiar	80% HLASM
	Used For	78% Development, 30% Maintenance, 13% Debugging
	Assembly More Difficult	64% Yes 38% Many low-level operations 31% Big picture obscured 25% Knowledge of underlying hardware/OS
	More Difficult	29% C/C++, 12% COBOL, 8% LISP
	Most Difficult Task	19% Testing, Debugging, Documentation 14% Understanding others' code, 10% Understanding new systems
	Most Time-Consuming Task	25% Testing, 20% Debugging, 20% Understanding others' code 15% Documentation, 10% Understanding new systems
Existing Tool Support	Primary Tool	68% Text editor, 12% HLASM Assembler
	Secondary Tool	60% Debugger
	Deficiencies	38% None 21% Text features (syntax highlighting, syntax checking) 13% Navigation within code
	Best Features	31% Data (register/var contents, memory/data flow) 19% Single step execution 13% Syntax highlighting 13% Trace or dump output
Browsing and Navigation	Beacons	76% Specific instructions, 16% Comments, Macros, Loops, None
	Task-Focused UI	36% No, 32% Yes, 32% Unsure
	Zoom By	29% Subroutines, 15% Do not have long modules (N/A) 12% Macros, CSECTs
	Additional	33%Follow links (branches, cross-refs, declarations)
Debugging	Features	Where is a particular subroutine/procedure invoked? 4.44 What are the arguments and results of a function? 4.76 How does control flow reach a particular location? 4.68 Where is a particular variable set, used or queried? 4.6 Where is a particular variable declared? 3.76 Where is a particular data object accessed? 4.28 What are the inputs and outputs of a module? 4.44
	Features Missing	32% Data flow concerns
	Mockup	Positive response
Control Flow	Static Concerns	None
	Dynamic Concerns	8% Most executed paths
	Forward CF	5.0/7 Useful
	Reversed CF	5.08/7 Useful, 90% Useful
	Data to Mine	24% Register values/mapping and memory usage (data) 12% Performance, 12% System, subroutine call statistics
Potential Tools	IDE Features	Syntax Highlighting 3.52 Syntax Checking 3.48 Code Completion 2.56 Search for References 3.96 Go to Declaration 3.88
	LegaSee	40% Useful, 12% Not Useful
	MapUI	31% Useful
	High-Level Split View	28% Not useful Confusion of using Assembly to develop in the High-Level PL
	UML Diagrams	7% state, 7% class
	Wish List	Pattern recognition Better macro processor, language and visualization Better debugging and breakpoints Better profiler to improve performance

Table A.1: Summary of Survey Results for Mainframe Respondents.

Section	Topic	Result
About the Participant	Dev Experience	79% 10+ Years
	Most Familiar PL	93% C/C++, 67% Java, 47% Assembly, 27% Python
	Favorite PL	47% C/C++, 40% Java, 20% Python
	Favorite Tool	47% IDA Pro, 40% Eclipse, 33% Visual Studio, 20% Text Editor
Assembly Experience	Writing	2.9/5
	Understanding	3.5/5
	Most Familiar	100% x86
	Used For	47% Malware understanding, 33% Program understanding 20% Reverse engineering
	Assembly More Difficult	80% Yes
		33% Many low-level operations, 20% Big picture obscured 13% Translate to high-level, reliance on conventions
	More Difficult	47% No, 33% Functional PLs, 7% Prolog
	Most Difficult Task	27% Control flow, 20% Data flow, 13% Deobfuscation, Decryption
	Most Time-Consuming Task	20% Locate behaviour 13% Control flow, Data flow, Deobfuscation, Decryption
Existing Tool Support	Primary Tool	87% IDA Pro
	Secondary Tool	33% Hex editors, 27% WinDbg, 20% IDA Pro plugins
	Deficiencies	20% Lack of integration 13% Instruction assistance, documentation 13% Conversion to high-level
	Best Features	20% IDA Pro graph view 13% IDA Pro extensibility, IDA Pro search patterns 13% Inspect and modify heap/registers/stack
Browsing and Navigation	Beacons	27% Function calls (control flow) 27% Data usage, 27% Coding conventions, 27% Function definitions
	Task-Focused UI	100% Yes
	Zoom By	40% Functions, 20% Modules
Debugging	Features	Where is a particular subroutine/procedure invoked? 4.80 What are the arguments and results of a function? 4.53 How does control flow reach a particular location? 4.60 Where is a particular variable set, used or queried? 4.60 Where is a particular variable declared? 3.53 Where is a particular data object accessed? 4.33 What are the inputs and outputs of a module? 4.13
	Features Missing	Varied (data flow, trace diffs, memory view, stepping backward, re-running system with reg/var values, access list to specific memory addresses, simulating execution statically, multi-application debugging, standard so all tools can communicate)
	Mockup	Negative response
Control Flow	Static Concerns	20% Loops and recursion
	Dynamic Concerns	7% Multi-threaded, compare traces, branch frequency
	Forward CF	6.38/7 Useful
	Reversed CF	6.15/7 Useful, 87% Useful
	Data to Mine	47% Call patterns 13% Compare traces, 12% How to reach execution points (jump conds)
Potential Tools	IDE Features	Syntax Highlighting 4.40 Syntax Checking 3.33 Code Completion 2.93 Search for References 4.73 Go to Declaration 4.60
	LegaSee	7% Useful
	MapUI	33% Useful, 20% Unsure
	High-Level Split View	85% Useful Already exists in VS Studio, Hex-Rays decompiler plugin
	UML Diagrams	33% state, 13% activity, 7% package
	Wish List	Better integration with other tools Meta-assembly to push back and forth with other tools Data flow, sequence viewer, pattern recognition, documentation, creating C from ASM to guess what a function is doing, omniscient debugging

Table A.2: Summary of Survey Results for Malware Respondents.

Appendix B

Script Used During the Nominal Group Session

Time	Action	Script
0 min Introduction	**SAY**	Hi, I'm Jennifer Baldwin from the University of Victoria in Canada. For my PhD in Computer Science, I am exploring how visualization and tool support for assembly language might be useful. My work is being funded by the Department of Defence and CA in Canada. Since you are experts in the area, we really value your experience and expertise in defining the issues. This is Alvin and I'll let him introduce himself. - ANU - Degree - Research Interest To get started, I'd like to collect the ethics forms that you were given yesterday.
	DO	Collect the ethics forms.
	SAY	This session should take no longer than 2 hours, including a 20 minute break. The aim is to discuss and **critically** rank all of the items from the exercise yesterday. If you come up with new ideas during the session, please add them to your list. Feel free to be **creative**. Does everyone have the blue pages? First of all, I'd like to go around the table and have everyone introduce themselves and tell us about your job. We also know from the survey that your teams are expertise-centered, so it would be great to hear about that, as well as your interests.

10 min Listing of Ideas	**SAY**	Now to begin the group exercise, we will go around the table and each person will share one item from their list at a time. At this time, please avoid discussion or talking out of turn. After all of the items are listed, we will have a discussion to clarify the items. If you have any new ideas then feel free to add them to your sheet. If you want to skip a turn, that is also fine.
	DO	Record word for word what each person says on the power point slide.
30 min Discussion of Ideas	**SAY**	We will now have a 30 minute discussion on all the ideas generated. Now is the time to ask for clarification or elaboration on an idea, or dispute or defend an item. You are also welcome to suggest new items during this time, but no items can be eliminated. We'll go through them item by item.
	DO	Announce each item on the list and ask what it means, or how people feel about it. Record any new ideas on the power point slide.
60 min Ranking to Select the "Top-Ten" Ideas	**SAY**	Now if everyone could take out their yellow sheet for preliminary ranking. You can see there are 10 spaces to be filled in. You can select 10 items that are the most important for you from all of the options. Then assign them a rank which is a numbering between 1 and 10, where 10 is the most important. Once you are finished, please turn it face down on the table and then you are free to take a break for about 20 minutes.
70 min Break	**DO**	Go around the table and transcribe and sum up the points from the ranking sheets onto the power point slides. Then reorder them on the slide based on the greatest number of points. Collect everyone from after their break.
90 min Discussion of Vote	**SAY**	We have reordered the items according to rank and you can see the score for them. We have also highlighted the top-ten. We will now have a free-for-all discussion about the nature and content of the top-ten. We would also like to hear how you feel about items that should have been included or excluded from this list.

110 min Re-Ranking and Rating Revised "Top-Ten" Items	**SAY**	Now if everyone could take out their green sheet for final ranking. Here you will again list the top ten items that you think are the most important. This may be the same ten, or feel free to modify which items are in your top ten. The ranking here is different in that 100 points will be given to the most important item. Every other item can have a value between 0 and 100. Two items can have the same ranking. Once you are finished, please hand in your sheets to me face down, and then we're all done!
	DO	Collect the green sheets from everyone and tally up the final scores based on the 0 to 100 ranking. **END CASE STUDY AT (START + 120 MIN)**

Appendix C

Activity-Based Elicitation Results

C.1 First Session at the Mainframe Group

Requirement Category	Issue	Description
Browsing and Navigation	XREF works on only 8 character long names	When there are more, search must be used, which only finds them one at a time in the code.
	Bookmarking lines of code	Have to create names "a", "b". If the name already exists, it is just overwritten.
	Lack of navigation	Need to scroll through many screens of code to look for the right spot.
Control Flow	Hard to find main task	
	Tools would need to support multi-threading	
Debugging	Timing issues were tricky	Timing dumps not useful because they are too complicated.
	Couldn't work out what was causing the cancel	Need some way to trap the event.
	XREF plus debugger to find the correct place to debug	Step through debugging might be helpful.
De-obfuscation	Redundant code makes the code confusing to read	Statements such as branching to the next address. Unnecessary since that code is next to be executed.
Documentation	Look up vendor error code in CA documentation	Not indexable online so need to download CA docs to search them. CA error code is then used to look up IBM Manual error code. Codes are OS version dependent.
	User prints off whole modules	The printoff is portable and more comfortable to look at (easier on the eyes). There are also sticky notes and writing on the pages. These written notes include variable names, addresses and error codes.
	The dump was scrolling off the page	There were so many errors, it did not fit. Need a way to condense it.
Source Control	Object module replacement	Overwrites whole module, have to be careful not to overwrite a change. Have to check prerequisite chain, and which fixes supersede others.

C.2 Second Session at the Mainframe Group

Requirement Category	Issue	Description
Browsing and Navigation	*temp is used as a TODO	Shows up only when you dig into the module you're interested in. Used *pdsman* to scan and find. Scan doesn't show the active module however.
	Switching terminal screens constantly	Need to scroll through many screens of code to look for the right spot. Kept many terminal screens open. Was hard to keep track of which showed the right code.
Build	Register usage	Waits for compile error to say that the register is in use.
	? at the start of lines	To ensure you get errors, but do no want to deal with the actual errors (stub error).
	Scanning software for changes he knows he has to make	Otherwise waits for compile errors. Compile errors would be better if they occurred during editing. Context aware correction suggestions (i.e. doesn't exist, did you mean...?). Calls out to code that does not exist anymore.
Debugging	No breakpoints in XDC	Puts code in to make it fail.
Documentation	IBM Principles of Hardware Manual	Useful to double check some things.
References	Code module - fan in, fan out	Wanted to know what module was being called dependent on the code and what code it depended on.
Source Editing	Tedious refactoring of modules	Splitting larger modules into smaller ones to use as templates. Templates are not useful, not maintained that much, but useful for people starting from scratch. Instead he uses something else he's working on, copies it and butchers it (side by side editing).
	Forgot to save the file	No alert was given.
	Code shortcuts	Stuff he does more than once.

Appendix D

Installing the AVA Framework

This appendix outlines how to install AVA for personal use. If you wish to further extend AVA, then please follow the development environment setup instructions instead.

D.1 Installing AVA

The binaries and source code for the AVA framework can be retrieved from GitHub at `https://github.com/jebaldwin/AVA`. Once the binaries have been downloaded, please copy the contents of the plugins folder to the plugins folder of the IDA Pro installation. You should have SequenceDiagramPlugin.plw under the plugins folder. The java folder with its contents should also reside in the same folder, for example C:/IDA/plugins/java/ava/ava.exe.

You may now run AVA by selecting "AVA Framework" under the plugins menu in IDA Pro. AVA will then launch and the connection with IDA Pro will be created automatically. If this plugin is selected from another running IDA Pro then it will connect with the single instance of AVA and not launch another.

D.2 Setting up the Development Environment

In order to develop for AVA, you will need to download the latest version of Eclipse for RCP and RAP developers. You will also need Visual C++ installed. Visual C++ 2008 Express Edition is recommended. AVA uses the Diver framework to create sequence diagrams, with minimal changes. The Diver framework with modifi-

cations is available at `https://github.com/jebaldwin/Diver`. You will need the *org.eclipse.zest.custom.sequence* project to compile AVA.

There are 4 source folders on the AVA GitHub repository (`https://github.com/jebaldwin/AVA`). These are:

IDAPlugin contains the code for the IDA Pro plugin

RCPApp contains the Eclipse code for the standalone RCP executable application

SocketModule contains the code for socket communication between the IDA Pro (C++) and Eclipse (Java) plugins

org.eclipse.zest.custom.sequence.assembly contains the Eclipse plugin code for the Assembly extension to Diver

In order to run the code, you will need to have two path variables set up as follows:

IDASDK = IDA Pro SDK directory

IDAPATH = IDA Pro installation directory

You will also need to have Java installed and included on your path.

When debugging or running the application from Visual Studio, you will see a message asking "Please specify the name of the executable file to be used for the debug session". Close Visual Studio and a vcproj file containing your computer and user name will have been created. The following lines in both the debug and release section must be changed to:

```
Command=''&quot;$(IDAPATH)\idag.exe&quot;''
WorkingDirectory=''&quot;$(IDAPATH)\plugins&quot;''
```

When debugging or running from Visual Studio, the SequenceDiagramPlugin.plw file will be automatically created and placed in the IDA Pro plugin directory. If you make changes to the Eclipse plugins, you can re-export the ava.exe file and place it within the java/ava folder under the IDA Pro plugins folder.

For easier debugging of the Eclipse plugin, you can establish the IDA Pro link without launching the executable. To do so, you must add -p:40010 to the Eclipse program arguments. This parameter is normally sent to the Java communication module when starting it from the IDA pro plugin. Next comment out the call to startUI() in the onRun function in IDAModule.cpp (line 249 at time of writing).

Finally there may be some dependencies that will need to be installed in your development Eclipse workbench to use all features of AVA. You will definitely need the following:

Zest The Eclipse Visualization Toolkit (`http://www.eclipse.org/gef/zest/`)
AJDT AspectJ Development Tools (`http://www.eclipse.org/ajdt/`)

Appendix E

Running the Mariposa Botnet with IDA Pro and Tracks

For more information on the Mariposa Botnet, please see Section 7.3.3. Additionally, [6] and [111] provide extensive analysis.

E.1 Environment Setup

The first step is to install the necessary tools. You will need to install Tracks as outlined in Appendix D.

You will also need to install the IDAStealth plugin (available at `http://newgre.net/idastealth`). Simply download the plugin to the plugins directory of your IDA Pro installation. Once installed, we will need to set up the correct preferences. To do, launch IDA Pro with any executable, then go to Edit -> Plugins -> IDA Stealth. The plugins menu is not visible unless an executable is loaded. You should enable everything on the two tabs "Stealth Techniques (1)" and "Stealth Techniques (2)". Set GetTickCount to 10. Your settings should look like Figure E.1.

Next, the **most important step** is to set a system restore point in Windows. This guide will provide information and address that will work while running Mariposa in Windows XP. These instructions will not work as written in other variants of Windows and will need to be altered accordingly. The restore point will allow us to revert to an uninfected state of the operating system. Set the system restore point by going to Start -> All Programs -> Accessories -> System Tools -> System Restore. Then click to create a restore point, go to next, then enter a restore point description and

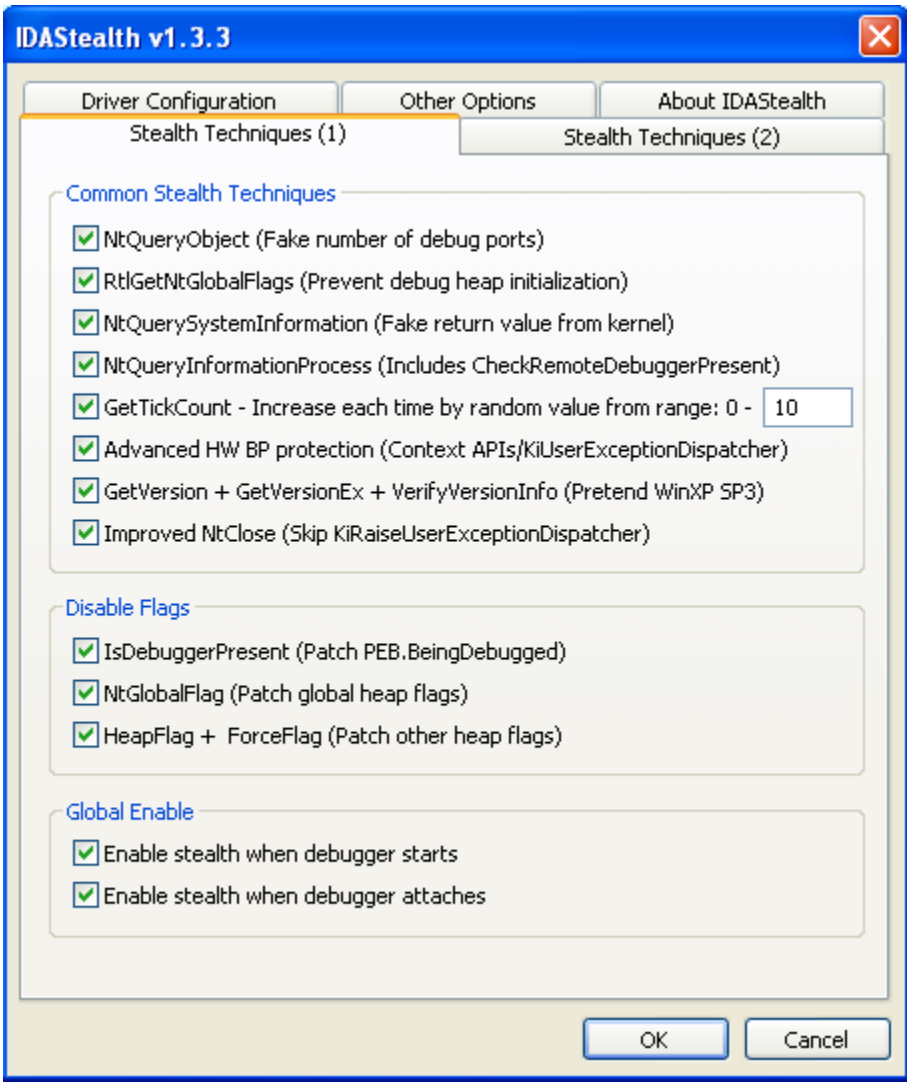

Figure E.1: IDAStealth Settings.

create.

Now that the restore point is created, you will need to temporarily disable your anti-virus protection. For some anti-virus tools this means uninstalling them. Others have the ability to momentarily suspend them.

E.2 Running Mariposa with Tracks

This guide will walk you through running Mariposa to create a dynamic control flow diagram of its behaviour with Tracks. This guide was written for Windows XP and used Butterfly_2009-08-26.exe for the botnet version.

E.2.1 IDA Pro Hot Keys

This is a list of important hot keys to be aware of, which we will use in our instructions. As a note, if you do not need to trace everything, then uncheck the appropriate tracing options in IDA Pro, as they will interfere with anti-debugging techniques.

```
F4 - run to cursor (put cursor on address listed and press F4)
F7 - step into
F8 - step over
F9 - continue (must have breakpoint set so you do not infect yourself)
Store Funcs - Memory Dump
Esc - goes back an instruction
G - Jump to address
Shift F2 - Brings up command window so you can ŞAnalyze AreaŤ
Ctrl E - set entry point
Ctrl F7 - step out
X - look for cross references
R - rename function
```

E.2.2 Running Mariposa

Before we begin analysis, DO NOT FORGET to create a System Restore Point!

Once the environment setup is completed, we will load the mariposa executable into IDA Pro and launch Tracks by selecting AVA Framework from the plugins menu. Once AVA is launched, a project called IDAPlugin will be created automatically and the static control flow file generated and placed inside. However there are no functions within IDA Pro since the code is packed, so this file will not be useful for us. Instead, we need to create a dynamic control flow diagram.

To do so, we will right click the IDAPlugin project, and go to New -> Other -> Assembly -> Debugging Diagram. The diagram will open with the user as the root of the diagram. Next we need to check are tracing options in IDA Pro. If we are not tracing then no calls will be sent to Tracks unless they are user-initiated (stepped through). If we experience an error, then toggle the following in IDA Pro:

```
Debugger -> Tracing -> Function Tracing
Debugger -> Tracing -> Instruction Tracing
```

First Decryption Layer and Obfuscation

We will need to create extra breakpoints after loops because Tracks automatically stops debugging for performance reasons once the loop/cycle count preference has been reached. However to resume tracing, Tracks needs to stop at another breakpoint to re-enable it. Create breakpoints at 41D469 and 41D482.

Next run to 41D469. Since this is the first breakpoint, either press F9 or start debugging to do so. There is a big cycle for decryption at loc_41D476 between the first two breakpoints. F7 or F9 to 41D482. A big loop for obfuscation exists at loc_-133FFA6, right before loc_41D047. F7 until 41D047. To make life easier, F4 over loop at loc_13FFA6.

Start of Anti-Debugging Traps

We need to enter a command by typing Shift F2. In the console window, enter - "AnalyzeArea(0x41D042, 0x41DDDD);" (without the quotes). Next we need to pass over over the big loop located at loc_41D057. This loop is used for obfuscation purposes. We do so by pressing F7 once, and then using F4 to get to 41D100. At this point, we may get an exception if this is the time running Mariposa. Select to pass the exception to the application. Then press F7 instead of F4. This HideDebugger.dll file comes from the IDA Stealth plugin. Now we use F4 to get to 41D127. There exists a big cycle for decryption at loc_41D137 between loc_41D14C.

Use G to jump to 41D2E0. Then press F4 to run to this address, followed by F7 to execute an instruction. Then press F8 to step over an instruction. At this point the second line should be 4100A7. Press F7.

Use G to jump to 410034. Then press F4 followed by F7.

Start of Second, Third and Fourth Decryption Layers

Now we are at loc_401000, which is the first location of data in this range, and goes to 0x415FB3.

Use G to jump to 40104B. Then press F4 followed by F7. We create a function here by pressing "p". This function will become sub_1320A0.

Use G to jump to 132323. Then press F4 followed by F7.

Code Injection

We create a function here at 1332D0 by pressing "p".

Use G to jump to 1334C0. Then press F4. The preparation for injection starts here at loc_1334C0. We therefore need to use F8 through here so we do not go into the APIs themselves.

Use G to jump to 135830. Press "p". Ensure that Tracks is not open at this stage otherwise we will get kicked out. Press F4. Use F7 to step through here, but use F8 over system calls so we do not enter into them.

Finding the process that Mariposa wants to inject into is done here at sub_135830. As a reminder, Mariposa injects into the "explorer.exe" process.

Use G to jump to G to 1357D0. Then press "p" and F4. The process of injection starts at loc_1357D0.

Use G to jump to 419e32. Press F4. You will now see "explorer.exe" on the screen. Go to View -> Open Subviews -> Hex dump.

At this point we want to prevent Mariposa from injecting into explorer.exe because the machine grinds to a halt, and further analysis is impossible. The process we change it to has to be a system process and has to have a name that is the same length. We have tried a few and found that winlogon.exe works best even though it requires restarting the machine. We have also tried uphclean.exe, mspdbsrv.exe and tfswctrl.exe which all have not worked.

Click on explorer.exe in the IDA Pro hex dump view, and press F2 to edit. Type in winlogon to replace explorer. Press F2 to commit the change. Then press Esc and Esc again to back out.

Use G to jump to 133765. Press F4 to run to here.

Now you will need to write down the address located in the stack view since it changes each time Mariposa runs. It should look like _ _ _ 0000 (EAX).

Use G to jump to 133CCF and then press F4.

Injected Code

In order to see the infected code running, we need to open another IDA Pro instance and press "Go" to work on our own.

In IDA Pro, go to Debugger -> Attach -> to local windows debugger. Now we pick the process we injected into. In our case, we pick winlogon.exe and then OK.

Use G to _ _ _ 1A20. These three empty spaces are the numbers from the

address that we wrote down in the previous step. This is the address at where the process will be injected.

Type Shift F2 to enter a command. Type "AnalyzeArea(0x_ _ _ 0000, 0x_ _-_ edfd);" in the console window (without quotes).

Use G to _ _ _1A20 (address as above), then press "p" to create a function. Set a breakpoint here at _ _ _ 1A20.

Then go to the Threads View -> Select All -> Right Click -> Suspend.

Now we go back to the other IDA Pro instance which is running Mariposa, and press F8 to create a remote thread. Then we go back to the IDA Pro instance running the process to be injected, and press F7 to see infected functionality.

Server communication starts one call before loc_1429BE9. Injecting the registry starts three API calls before loc_1431615, and injected functionality starts two API calls before loc_1431A9D.

After Analysis

Once our analysis has been complete, it is important that we reset our system. First close IDA Pro and do not pack the database, and do not save.

The most important step now is to reset windows. We do so by going to Start -> All Programs -> Accessories -> System Tools -> System Restore. Then select to restore my computer to an earlier time -> Next. Choose the date and description from the restore point created at the beginning and press next and confirm. Your system will restore and restart.

To confirm that we are not infected, we will go to Start -> Run -> and then type regedit and press ok. Select My Computer. Search by pressing Ctrl-f and typing "Taskman", check Values and Match Whole String. If found, delete it and wait a few seconds. Then press F5 to refresh. If it shows up again, then the machine is infected.

Appendix F

Ethics Approval

This appendix provides the renewal approval certificate for ethics protocol number 10-241 from the University of Victoria. Ethics was originally approved on June 22, 2010.

University of Victoria

Human Research Ethics Board
Office of Research Services
Administrative Services Building
PO Box 1700 STN CSC
Victoria British Columbia V8W 2Y2 Canada
Tel 250-472-4545, Fax 250-721-8960
Email ethics@uvic.ca Web www.research.uvic.ca

Certificate of Renewed Approval

PRINCIPAL INVESTIGATOR:	**Jennifer Baldwin**	**ETHICS PROTOCOL NUMBER** Minimal Risk - Delegated	**10-241**
UVic STATUS:	**Ph.D. Student**	ORIGINAL APPROVAL DATE:	22-Jun-10
UVic DEPARTMENT:	**COSI**	RENEWED ON:	13-May-13
SUPERVISOR:	**Dr. Yvonne Coady**	APPROVAL EXPIRY DATE:	21-Jun-14

PROJECT TITLE: **Assembly Code Visualization and Analysis**

RESEARCH TEAM MEMBERS: Co-investigator: Laura mcLeod (UVic)

DECLARED PROJECT FUNDING: **NSERC (2010)**

CONDITIONS OF APPROVAL

This Certificate of Approval is valid for the above term provided there is no change in the protocol.

Modifications
To make any changes to the approved research procedures in your study, please submit a "Request for Modification" form. You must receive ethics approval before proceeding with your modified protocol.

Renewals
Your ethics approval must be current for the period during which you are recruiting participants or collecting data. To renew your protocol, please submit a "Request for Renewal" form before the expiry date on your certificate. You will be sent an emailed reminder prompting you to renew your protocol about six weeks before your expiry date.

Project Closures
When you have completed all data collection activities and will have no further contact with participants, please notify the Human Research Ethics Board by submitting a "Notice of Project Completion" form.

Certification

This certifies that the UVic Human Research Ethics Board has examined this research protocol and concluded that, in all respects, the proposed research meets the appropriate standards of ethics as outlined by the University of Victoria Research Regulations Involving Human Participants.

[redacted]

Dr. Rachael Scarth
Associate Vice-President, Research

Certificate Issued On: 13-May-13

10-241 Baldwin, Jennifer