Visualization and Analysis of Assembly Code in an Integrated Comprehension Environment

by

Dean W. Pucsek
B.Eng., Carleton University, 2008

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

MASTER OF SCIENCE

in the Department of Computer Science

Visualization and Analysis of Assembly Code in an Integrated Comprehension Environment

by

Dean W. Pucsek
B.Eng., Carleton University, 2008

Supervisory Committee

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

Dr. H. Muller, Departmental Member
(Department of Computer Science)

Supervisory Committee

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

Dr. H. Muller, Departmental Member
(Department of Computer Science)

ABSTRACT

Computing has reached a point where it is visible in almost every aspect of one's daily activities. Consider, for example, a typical household. There will be a desktop computer, game console, tablet computer, and smartphones built using different types of processors and instruction sets. To support the pervasive and heterogeneous nature of computing there has been many advances in programming languages, hardware features, and increasingly complex software systems. One task that is shared by all people who work with software is the need to develop a concrete understanding of foreign code so that tasks such as bug fixing, feature implementation, and security audits can be conducted. To do this tools are needed to help present the code in a manner that is conducive to comprehension and allows for knowledge to be transferred. Current tools for program comprehension are aimed at high-level languages and do not provide a platform for assembly code comprehension that is extensible both in terms of the supported environment as well as the supported analysis.

This thesis presents *ICE*, an *Integrated Comprehension Environment*, that is developed to support comprehension of assembly code while remaining extensible. ICE is designed to receive data from external tools, such as disassemblers and debuggers, which is then presented in a series of visualizations: Cartographer, Tracks, and a Control Flow Graph. Cartographer displays an interactive function call graph while Tracks displays a navigable sequence diagram. Support for new visualizations is provided through the extensible implementation enabling analysts to develop visualizations tailored to their needs. Evaluation of ICE is completed through a series of

case studies that demonstrate different aspects of ICE relative to currently available tools.

Contents

List of Tables

List of Figures

List of Listings

ACKNOWLEDGEMENTS

I would like to thank:

Mark, Sharon, Matthew, Blake, Kerry, Chris Aylard, and Bailey Adamson,
for supporting me in all I do.

Yvonne Coady,
for mentoring, support, encouragement, and patience.

Defence Research and Development Canada,
for financial support throughout this endeavour.

DEDICATION

UVic Vikes Rowing and Rowing Canada.

It is the discipline and commitment I've learned on the water that has enabled me to achieve my goals off the water.

Chapter 1

Introduction and Related Work

Computing has reached a point where it is visible in almost every aspect of one's daily activities. Consider, for example, a typical household. There will be a desktop computer, game console, tablet computer, and smartphones built using a different types of processors and instruction sets. To support the pervasive and heterogeneous nature of computing there has been many advances in programming languages, hardware features, and increasingly complex software systems. For example, modern processors now include hardware virtualization to better support cloud computing; software is now typically written in high-level languages that enable programmers to more easily express their ideas; and, technologies such as multi-threading are now commonplace in software.

In conjunction with all of the new software being developed there is also the issues of maintaining legacy software and performing security audits on existing software. Legacy software continues to be used, such as in some mainframes, and for one reason or another must continue to be maintained. Security audits are becoming increasingly important as people enlist computers in areas such as health care, military, and critical infrastructure.

One task that is shared by all people who work with software is the need to develop a concrete understanding of foreign code so that tasks such as bug fixing, feature implementation, and security audits can be conducted. To do this tools are needed to help present the code in a manner that is conducive to comprehension and allows for knowledge to be transferred.

1.1 Program Comprehension

Program comprehension is the task of developing an understanding of a particular piece of software; the understanding can be either functional—*how* the software works—or holistic—*what* the software does [63, 50]. The need for program comprehension is seen in all areas of computing such as: security audits, development [48, 45], and educational purposes [50].

Despite this need for program comprehension each group of people that interacts with software may have access to different types of information (e.g. source code, access to developers, documentation) and, as a result, a different set of tools available.

A software engineer—and student—typically has access to a wealth of information and is able to leverage a wide variety of tools [16, 17, 24, 62, 15, 54, 41], including those that specifically target the language in use. Conversely, a reverse engineer is faced with a lack of information available and must rely on tools such as assembly-level debuggers [43, 28] and disassemblers [33, 32, 21]. It is these differences in information and tool availability, as well as the differences in the environment that lead to several issues for reverse engineers when faced with the task of understanding the implementation and functional details of a program.

Of the many groups that interact with software, reverse engineers are tasked with the job of taking already written code—usually in binary form—and developing an understanding of both the functional and holistic elements. In order to develop this understanding reverse engineers must work with assembly code which leads to three primary issues.

1. *Information Overload:* Reverse engineers must deal with an extremely large number of assembly instructions since each high-level statement is translated into at least one assembly instruction, if not more.

2. *Information Loss:* Assembly code does not include information such as variable types, function names, and structure definitions that, in other situations, can provide a great deal of insight.

3. *Tool Support:* Few tools exist to assist a reverse engineer in understanding assembly code. Furthermore, available tools tend to be centred around a particular type of assembly code and lack visual components to assist comprehension.

1.2 Assembly Code

Even with an extraordinarily high affinity towards high-level languages, assembly code is still prevalent and widely used [59]. For example, assembly code is still widely used in mainframe programming.

Due to the fact that assembly code is a human-readable representation of a processors machine instructions the syntax and semantics are not necessarily transferable. This unbreakable tie between assembly code and processors gives rise to the need for flexibility in approaches to program comprehension.

To help illustrate the wide variety of assembly code Listing 1.1 shows an implementation of a string length routine in x86; the architecture found in most desktop computers.

Listing 1.1: Implementation of string length in x86

```
       push   ebp                ; save previous stack pointer
       mov    ebp, esp           ; adjust stack to current frame
       mov    eax, [esp+4]       ; get pointer to string
next:  cmp    byte ptr [eax], 0  ; is null terminator?
       je     done               ; if terminator, jump
       inc    eax                ; else, increment string pointer
       jmp    next               ; and loop
done:  sub    eax, [esp+4]       ; subtract start of string pointer
                                  ; from end of string pointer
       leave                     ; release stack frame
       ret                       ; return to caller
```

Similarly, Listing 1.2 is the same string length algorithm implemented in ARM, found in most smartphones and tablet computing devices.

Listing 1.2: Implementation of string length in ARM

```
       stmfd sp!, {r1, r2, lr}   @ preserve caller values of r1 and
                                 @ r2, return address on the stack
       mov   r2, r0              @ keep copy of start string pointer
next:  ldrb  r1, [r0], #0x1      @ place current character in r1,
                                 @ increment string pointer by 1
       cmp   r1, #0x0            @ is current the null terminator?
       beq   done                @ if yes, go to done
       bal   next                @ if no, continue loop
done:  sub   r0, r0, r2          @ subtract start of string pointer
                                 @ from end of string pointer
```

```
        ldmfd sp!, {r1, r2, pc}    @ restore caller values of r1 and
                                   @ r2, put return address in
                                   @ program counter
```

Although assembly code may be encountered from numerous sources one of the most common is disassembly.

1.2.1 Disassembly

A disassembler takes as input a binary program and returns as output a representation, usually textual, of the machine code. The need for disassembly is two-fold. On one hand, a reverse engineer may only have access to a binary and therefore must disassemble it in order to have a starting point for program comprehension. On the other hand, a reverse engineer may have access to the source code in addition to a binary and will disassemble the binary in order to verify that the source code provided could have generated the binary [4].

There are many tools [33, 32, 2, 1] and techniques [36, 13, 44] available to disassemble code; however, IDA Pro [33] is generally accepted as the industry standard. IDA Pro boasts a long history coupled with support for multiple binary file formats, supported operating systems, and supported instruction sets.

As stated in Section 1.1, one of the major drawbacks to program comprehension based on assembly code is the sheer amount of code to be processed. The challenge here is largely cognitive in that it is extremely difficult for a reverse engineer to keep track of all pertinent details while developing a complete understanding of the program [8]. Furthermore, in malicious environments it is not reasonable to assume that the disassembly is correct due to techniques employed by malware authors [64, 26].

1.2.2 Decompilation

An approach to alleviating the cumbersome nature of assembly code is to *decompile*, or translate, it into a high-level language or pseudo-language. While the techniques [13, 14] used to decompile are outside the scope of this thesis, it is important to note some of the drawbacks of this approach.

The first, and foremost, drawback to decompilation is that it is an undecidable problem [61] and is therefore not possible in all cases. Current decompilers work

around this by limiting themselves to specific classes of code (e.g. strictly conforming C code) or making assumptions about the resulting code. However, this limitation is magnified when one considers code outside the class the technique was developed for and especially when faced with potentially malicious code.

A second drawback to decompilation is that it is difficult to conclusively identify the data type or high-level control flow structure used [61]. Consider how a decompiler might differentiate between a 32-bit integer and a 32-bit pointer value. In this case the correct interpretation depends on the context the value is used in which is not necessarily possible to determine during decompilation.

A third drawback to decompilation is that it is not well suited to the object-oriented, and dynamic, nature of languages such as C++ [27]. In the case of C++ some of the issues that are encountered are related to reconstructing the class hierarchy, identifying and associating member functions, and reconstructing exception handling code blocks.

1.3 Visualizations

Through stakeholders, one aspect that was identified as necessary for a program comprehension environment suited for assembly code is the usage of visualizations to display different aspects of the code being analyzed. At the core of each visualization is a type of graph designed to show some specific type of information; for example, a *function call graph* displays the relationship between the caller and callee. The following is a brief survey of commonly used graphs and the type of information they focus on.

A *call graph* is a directed graph (Figure 1.3) that represents the caller-callee relationship in functions [49, 11]. Since every modern programming language supports the notion of a function the call graph is language and paradigm agnostic [29]. From the call graph a reverse engineer is able to form an understanding of the structure of the program and, provided accurate function names are available in the binary, able to deduce the action carried out in each function.

Sequence diagrams are a visualization that depict the interactions between objects in the sequential order they occur. Figure 1.3[1] shows a sample sequence diagram in which the events required for a student to register in a class are examined. Sequence

[1]Image source: `http://www.ibm.com/developerworks/rational/library/3101.html`

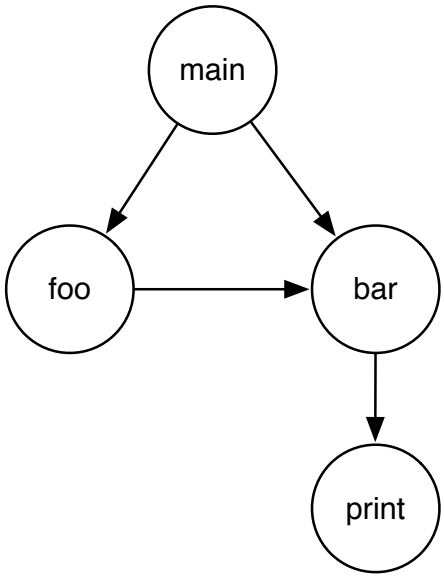

Figure 1.1: A sample call graph

diagrams were born in the *Unified Model Language* [34] and tend to be used when describing object-oriented code bases.

Continuing with the effort to identify relationships in program components are *software terrain maps* [19]. Software terrain maps (Figure 1.3) provide a spatial representation of relationships between functions. One notable feature of software terrain maps is that functions are placed in the map such that their size is indicative of the function size and location depicts the relationship with surrounding functions.

Despite the amount of information and understanding that can be obtained at a functional (or global) level, at times it is necessary to delve into the implementation details of a single function. For this task, a *control flow graph (CFG)* [3] is commonly used. Control flow graphs operate on basic blocks, a set of assembly instructions that has at most one entrance and one exit, and provide insight into the paths that can be taken within a function as well as how the various paths relate to each other.

Finally, there continues to be new graphing and diagramming approaches created as analysts better understand the type of information they are after. Tree maps and thread graphs [60] aim to give insight into the behavioural aspects of a program; where as distribution maps [22] aim to visualize the properties of a software system identified by a human analyst.

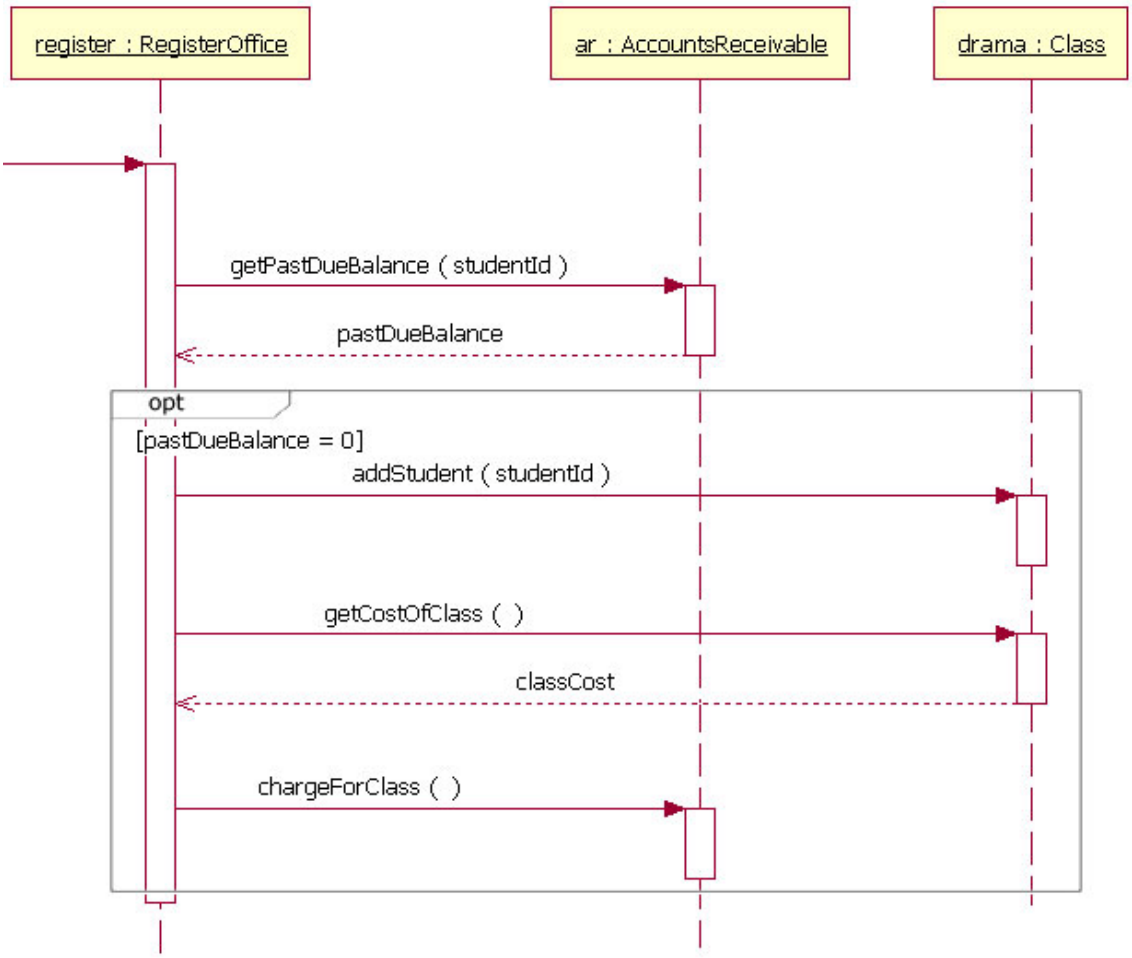

Figure 1.2: A sample sequence diagram.

1.4 Foundations for Comprehension

The primary goal of this thesis is to improve comprehension of assembly code through visualizations and analyses. Before developing the prototype discussed in the following chapters a survey of related approaches was conducted. We found that existing solutions can be classified in one of two categories: binary-based frameworks and intermediate language-based frameworks.

1.4.1 Binary-Based Frameworks

One approach to developing tools for program comprehension is to build a framework around a specific type of binary. In this approach the interface for tools has specific

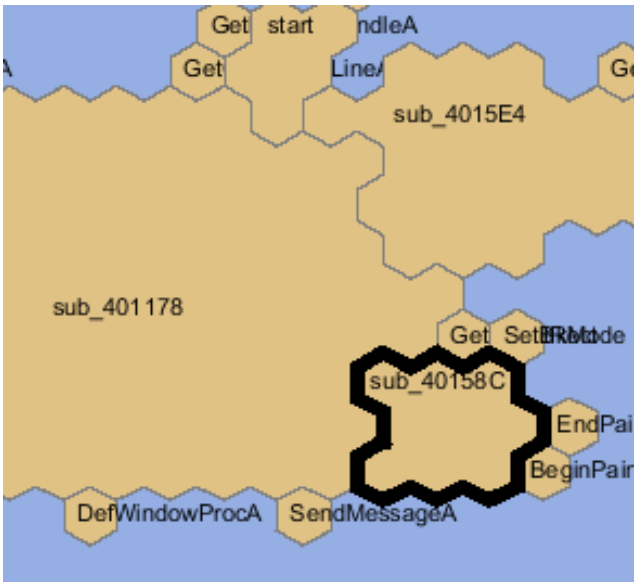

Figure 1.3: A software terrain map

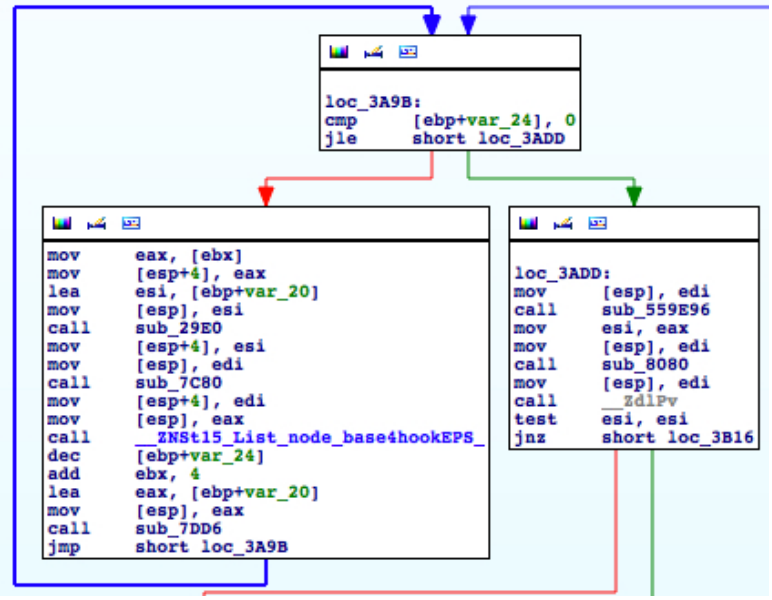

Figure 1.4: Control flow graph showing a for-loop.

knowledge about the binary being examined. For example, if a Portable Executable (PE) [2] binary that contains code for an Intel 32-bit system must be analyzed then the framework would have specific knowledge of this type of binary and know how to access specific information such as the binary headers.

[2]Portable Executable binaries are commonly found on Microsoft Windows systems.

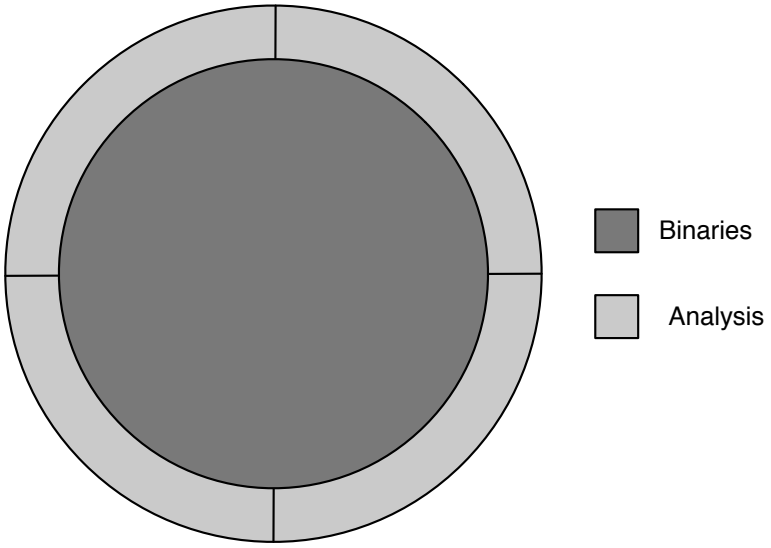

Figure 1.5: Conceptual representation of a binary-based framework

As seen in Figure 1.5, the binary is a central component and the analyses are developed around it. The benefit of this approach is that the coupling between the interface used by analyses and the binary is extremely tight enabling for very specific information to be extracted from the binary. Moreover, analyses are tightly integrated with each other enabling common functionality to be shared.

The tight integration found in a framework comes at the cost of having to develop a separate framework for each type of binary being analyzed. Therefore, if an analyst needed to examine a Mach-O [3] executable containing Intel 64-bit code an entirely new framework must be developed.

A summary of numerous binary-based frameworks follows.

IDA Pro

IDA Pro [30] is an industry-standard interactive disassembler. It has been designed such that it is able to integrate a suite of builtin analysis tools with those provided by third-parties in order to provide an extensible general-purpose binary analysis framework. IDA Pro provides this functionality through a traditional plugin architecture and a well-defined API that allows third-party developers to produce task-specific analysis algorithms [5]. The primary strength of IDA Pro is its disassembler which supports a multitude of instruction sets. Figure 1.6 shows a typical IDA Pro session. [4]

[3] Mach-O binaries are commonly found on Mac OS X systems.

[4] Image is from https://www.hex-rays.com/products/ida/index.shtml

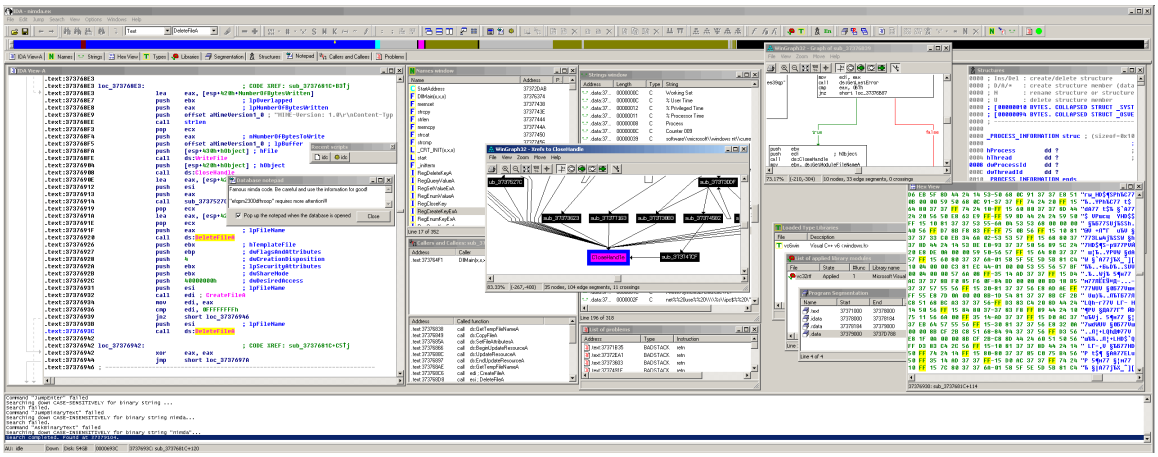

Figure 1.6: IDA Pro during a typical reverse engineering session

BitBlaze

BitBlaze [53] is an open-source project consisting of two integrated binary analysis frameworks (one static, one dynamic), and a mixed concrete and symbolic execution engine. BitBlaze was developed at UC Berkeley and aims to provide a better understanding of software through a fusion of static and dynamic analysis. It supports both static and dynamic analysis of 32-bit x86 binaries. All tools produce textual output only through a command-line interface. The analysis tools are dependent on the intermediate language, which is generated using several third-party tools including an emulator called QEMU [9]; a recent study names QEMU as one of four emulators that allegedly unfaithfully emulates certain instructions [39]. The static analysis framework—Vine—can be extended by building tools on top of the Vine IL. In the dynamic analysis framework, extensibility is achieved through plugins to the BitBlaze emulator, TEMU, which is based on QEMU. Shortly after BitBlaze was developed two members of the research team released another framework, BAP (Binary Analysis Platform) [18].

BinNavi

BinNavi [66], Figure 1.7 [5], is a binary analysis framework produced by Zynamics and is specifically designed to facilitate vulnerability detection in executables. BinNavi makes heavy usage of graph-based visualizations during analysis and currently supports x66, PowerPC, and ARM code. As with IDA Pro, BinNavi provides ex-

[5]BinNavi screenshot from http://www.zynamics.com/binnavi.html

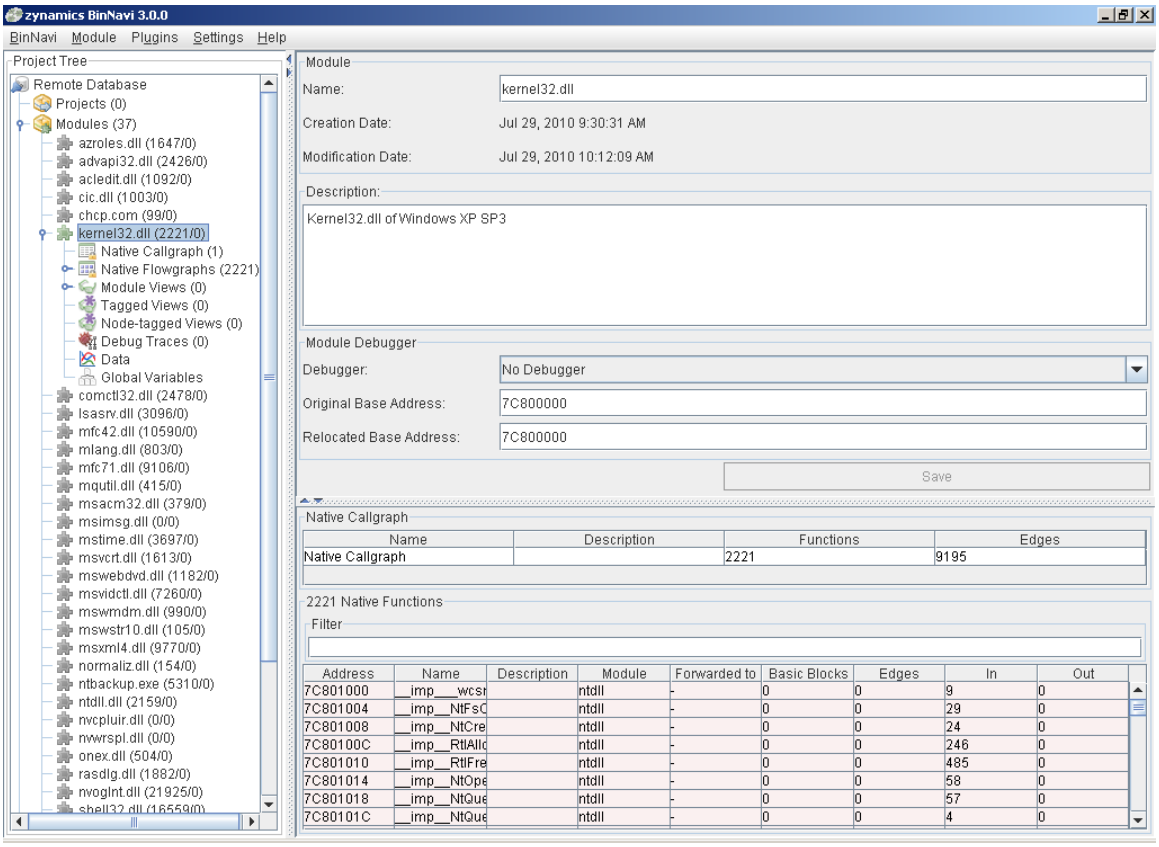

Figure 1.7: Main window of BinNavi

tensibility using a plug-in architecture and is platform-independent. BinNavi can be either used as a standalone tool or leverage the disassembling capabilities of IDA Pro. If IDA Pro is used then the disassembly data must be exported to BinNavi using an IDA Pro plugin.

HERO

HERO (Hybrid sEcurity extension of binaRy translatiOn) [31] is a promising academic framework that claims to support an efficient combination of static and dynamic binary analysis methods. HERO is designed specifically for malware analysis and it claims to be entirely self-contained; that is to say, it does not rely on any third-party components. The intermediate language used within HERO, according to the paper, is formally specified however no details are given.

Valgrind

Valgrind [42] is a robust, well-established framework that provides support for both static and dynamic analysis. It runs on multiple flavours of Linux and Mac OS X, with plans to extend to more operating systems. It uses two intermediate languages: VEX and UCode, a RISC-like language. Each instruction is translated individually and independently and unsupported instructions are inserted as comments to preserve dynamic analysis functionality.

1.4.2 Intermediate Language-Based Frameworks

The second approach, an intermediate language-based framework, focuses on encapsulating common aspects of binaries in an intermediate language. This approach enables for a potentially limitless number of binary file format and instruction set combinations as well as enables analyses to be written once for the intermediate language. However, an intermediate language-based approach does require a translator to be written for each instruction set in order to be supported in the framework. Figure 1.8 depicts a schematic of this approach.

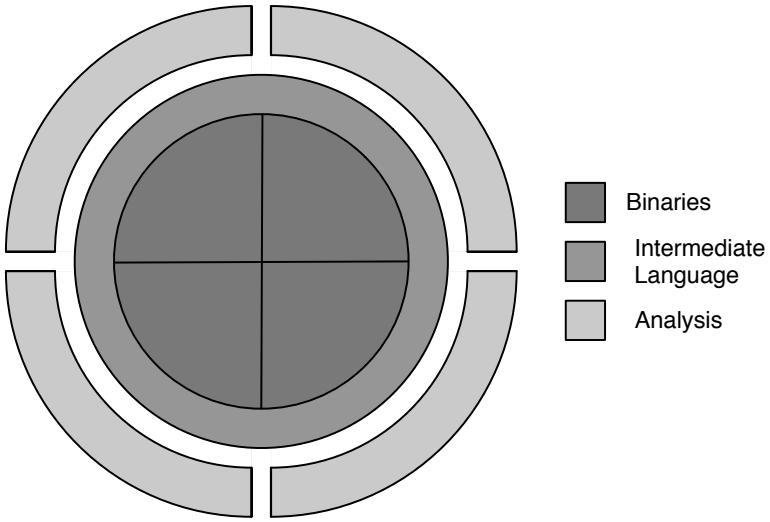

Figure 1.8: Conceptual representation of a intermediate language

The primary advantage of this approach is the ability to support a range of instruction sets; however, this approach raises numerous challenges as well. First, since each analysis tool is based upon an intermediate language it is not necessarily integrated with other tools leading to a potentially significant lack of code re-use and the

potential for incompatibilities between analysis tools. Second, the intermediate language must be designed to satisfactorily encapsulate various aspects of an instruction set and the associated processor. For example, the intermediate language should be able to encapsulate the notion of processor status flags which may differ across processors. Additionally, there is potential for complications to arise when attempting to incorporate functionality such as SIMD and hardware virtualization.

A summary of intermediate language-based frameworks follows.

LLVM Code

LLVM [37] is an open-source project that was initially designed as a framework for compiler construction but has now evolved to provide a collection of tools and libraries including debuggers, disassemblers, and high-level language parsers. At the core of LLVM is a language that can be used in three forms: an in-memory compiler intermediate representation, an on-disk byte-code representation, and a human readable representation. As noted in the LLVM language reference [38] aims to be a "universal IR" that is low-level yet capable of mapping to high-level constructs. The LLVM language contains a fully specified instruction set as well as many other functions and data types relevant to program comprehension. In particular, it defines a mechanism to attach metadata to any translation and supports FPU, MMX, and SSE instructions and data types. Extensibility in LLVM is almost unlimited, as mechanisms are provided to add instructions, types, and intrinsics[6]. Listing 1.3 shows the LLVM code output when compiling the implementation of string length presented in Section 1.2.

Listing 1.3: Implementation of string length in LLVM

```
define i32 @length(i8* %str1) nounwind uwtable ssp {
  %1 = alloca i8*, align 8
  %str2 = alloca i8*, align 8
  store i8* %str1, i8** %1, align 8
  %2 = load i8** %1, align 8
  store i8* %2, i8** %str2, align 8
  br label %3

; <label>:3                                       ; preds = %8, %0
```

[6]An intrinsic is a compiler specific, highly optimized function provided by a language where the underlying optimal implementation is handled by the compiler. The compiler has knowledge of the intrinsic function and can integrate based on the situation or program circumstances.

```
  %4 = load i8** %1, align 8
  %5 = load i8* %4, align 1
  %6 = sext i8 %5 to i32
  %7 = icmp ne i32 %6, 0
  br i1 %7, label %8, label %11

; <label>:8                                          ; preds = %3
  %9 = load i8** %1, align 8
  %10 = getelementptr inbounds i8* %9, i32 1
  store i8* %10, i8** %1, align 8
  br label %3

; <label>:11                                         ; preds = %3
  %12 = load i8** %1, align 8
  %13 = load i8** %str2, align 8
  %14 = ptrtoint i8* %12 to i64
  %15 = ptrtoint i8* %13 to i64
  %16 = sub i64 %14, %15
  %17 = trunc i64 %16 to i32
  ret i32 %17
}
```

Reverse Engineering Intermediate Language (REIL)

REIL (Reverse Engineering Intermediate Language) [23] provides a platform-independent intermediate language of disassembled code for static analysis. REIL uses a side-effect-free, RISC-style intermediate language consisting of 17 instructions. Each instruction contains exactly three operands (some instructions have operands of type `<empty>`), making the operation of individual instructions easier to understand. REIL instructions can easily be traced back to the original assembly instruction since the address of REIL instruction is the address of the original instruction with an offset value appended to the end. Unrecognized source instructions are essentially ignored by translating them into the `UNKN` instruction, a variant of a `NOP` instruction, rendering dynamic analysis unfeasible. Finally, there is no mechanism to extend the REIL instruction set and REIL does not support instruction set extensions such as SSE, virtualization, and floating point operations. Listing 1.4 shows the REIL implementation of the string length routine discussed in Section 1.2.

Listing 1.4: Implementation of string length in REIL

```
strlen1:
0x00100    str  esp,   , t1
0x00101    add  t1,  4, t2
0x00102    ldm  t2,   , t3
0x00103    str  t3,   , eax

0x00200    ldm  eax,   , t4
0x00201    bisz t4,   , t5

0x00300    jcc  t5,   , 0x00600

0x00400    add  eax,  1, t6
0x00401    str  t6,   , eax

0x00500    jcc  1,   , 0x00200

0x00600    add  esp,  4, t7
0x00601    ldm  t7,   , t8
0x00602    str  t8,   , eax
0x00603    sub  t8, t9, eax

0x00700    str  esp,   , t10
0x00701    sub  t10,  4, t11
0x00702    ldm  t11,   , eip
0x00703    jcc  1,   , eip
```

Static Analysis Intermediate Language (SAIL)

SAIL (Static Analysis Intermediate Language) [20] is an open-source project that translates C—or C++—code into two complimentary intermediate languages: a high-level language, and a low-level language. The high-level language retains constructs from the original source code where as the low-level language is source code independent and is more amenable to static analysis.

1.5 Requirements for Comprehension

A study completed by a colleague designed to understand the needs of developers and analysts who work with assembly code on a regular basis revealed many common

requirements for tools in the domain of program comprehension [7]. As a proof-of-concept, this work focuses on the following subset of requirements identified by the analysts in that study:

1. *Multiple Executables:* Their disassembler cannot disassemble more than one executable file at a time (e.g. DLL libraries) and link between them.

2. *Map of Analysis:* It is easy to get lost when going deeper into the code—hard to track where the exploration started and how a deeper point was arrived at.

3. *Tagging:* There is no tagging mechanism for assembly where, for example, one could tag a global variable and see where it comes from.

4. *Cross Reference Mechanism:* Lack of a cross reference mechanism between a given function in an executable file to a DLL.

The requirements were ranked in the study and, although the above are a subset, the first listed (providing support for multiple executables) received the highest rank out of the total 15 requirements. The second and third in the above list were fourth and fifth respectively in the final ranking while the fourth listed was number 12.

1.6 Thesis Statement

The goal of this work is to explore the feasibility of applying principles from high-level program comprehension tools to low-level codebases. The design and implementation of ICE demonstrates that it is possible to build an extensible framework for interactive visualizations that is flexible in terms of the data acquisition.

1.7 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 introduces *ICE*, an *Integrated Comprehension Environment*, and discusses characteristics of the design as well as the implementation. Chapter 3 then discusses three case studies that investigate the ability of ICE to assist an analyst with program comprehension. A discussion and evaluation of ICE is then presented in Chapter 4 followed by future work and a conclusion presented in Chapter 5.

1.8 Summary

In this chapter the notion of program comprehension was introduced along with several approaches. The cognitive disadvantage of assembly code was raised as a core issue; decompilation and visualization where identified as current solutions to the problem.

Chapter 2

ICE: Evolution, Design, and Implementation

Given the challenges outlined in Chapter 1—and further developed in [7]—my thesis proposes *ICE*, an *Integrated Comprehension Environment*, as a framework for program comprehension that is based on extensibility and modularity. This chapter delves into the evolution, design, and prototype implementation of ICE as well as the guiding principles that helped shape the overall development process.

2.1 Guiding Principles

Early in the project several guidelines were established that helped steer the development process and work completed. The guidelines are:

1. Instruction set independent
2. Extensible visualizations
3. Flexible data acquisition
4. Operating system independent

The relevance and impact of each guideline on the development process is varied; however, the unifying concept between all four guidelines is the need to remove as many limitations as possible while providing a cohesive solution for program comprehension.

Instruction set independent. The first guideline conceived was the need for ICE to be instruction set and binary file format independent. This guideline came about because of the observation that there is an ever increasing number of instruction sets available on the market; especially as new computing paradigms become ubiquitous. A motivating example of this guideline comes from the domain of embedded devices as processors become better tailored to specific tasks. In this case, the ARM processor has become commonplace with the rise in smartphone and tablet computing devices. Similarly, the burgeoning domain of general purpose GPU computing has brought new, GPU specific, instruction sets into the main stream. The impact of this guideline on the development of ICE is that there is a need to better understand what differences exist between various instruction sets and how these differences could be reconciled. This guideline was a large motivating factor for the investigation of frameworks and intermediate languages discussed in Section 1.4.2.

Extensible visualizations. The next guideline was that the visualization infrastructure provided by ICE must be extensible. The need for this guideline lies in the fact that (1) not all analysts will be interested in the same set of visualizations, and (2) in order to foster further research in program comprehension it is important to enable a researcher to develop and explore custom visualizations. As a result of this guideline ICE was developed around the *Model-View-Controller (MVC)* [46] paradigm in order to provide a clear boundary between the data available for visualizations and the visualizations themselves.

Flexible data acquisition. Since ICE currently does not parse or disassemble any binaries this guideline was put in place to allow ICE to be able to accept data from a wide variety of sources. For example, it may be necessary for an analyst to retrieve data from a disassembler while concurrently retrieving data from a debugger. Furthermore, this guideline also serves to support the need for ICE to be capable of working with multiple binaries simultaneously. In addition to the need for an extensible visualization infrastructure this guideline was another primary motivator to leverage the MVC paradigm. This guideline also shaped a large portion of the communication mechanism used between ICE and the data sources.

Operating system independent. The final guideline that helped to shape the development process of ICE was that ICE should be operating system independent.

This guideline was largely founded in the growth of operating systems other than Microsoft Windows and the idea that to help future-proof ICE it should not mandate a specific operating system for the analyst to use. The impact of this guideline on ICE was that all supporting libraries used in the prototype must also be operating system independent; additionally, this guideline had a large influence on the selection of programming language and environment.

2.2 Evolution of ICE

Before delving into the design and implementation of ICE it is important to understand the process that lead to its inception. Following the investigation of frameworks and intermediate languages (Section 1.4.2) we decided that, to provide a reasonable solution to the problem of assembly code analysis, an approach that leveraged both frameworks and intermediate languages was required.

The need for a hybrid solution follows from the fact that multiple instruction sets need to be supported in order to support the wide variety of modern electronics. Through a hybrid solution it would be possible to write the vast majority of analyses and visualizations against a single intermediate language while providing access to arbitrary data in a binary through an API.

2.2.1 REIL Translator and Simulator

Due to the simplicity of leveraging a pre-existing intermediate language the first step taken after the initial investigation into intermediate languages was to develop a translator and simulator for REIL. Both of which were written in Python.

The translator took as input a disassembly of 32-bit Intel code and produced as output semantically equivalent REIL code. As an example, given the instruction `mov eax, [esp-4]` the REIL code shown in Listing 2.1 would be generated.

Listing 2.1: Sample REIL Code

```
sub t1, esp, 4
ldm t2, t1
mov eax, t2
```

In this example, the value 4 is subtracted from `esp` (the stack pointer), then the value stored in memory at that location is loaded into `t2`, and finally that value is moved into `eax`.

With nearly 600 instructions in the Intel instruction set, the translator did not implement each instruction. Instead it focused on a core set that are commonly used by compilers. The translator was tested using various implementations of functions that computed the length of a null-terminated string.

As a proof-of-concept, a colleague developed a simulator that was able to evaluate the REIL code generated by the translator. The simulator allowed for inspection of the memory as well as registers in use. The veracity of the simulator was validated by comparing the results of the string length computations to values computed using the standard `strlen()` function available in C.

2.2.2 Rails

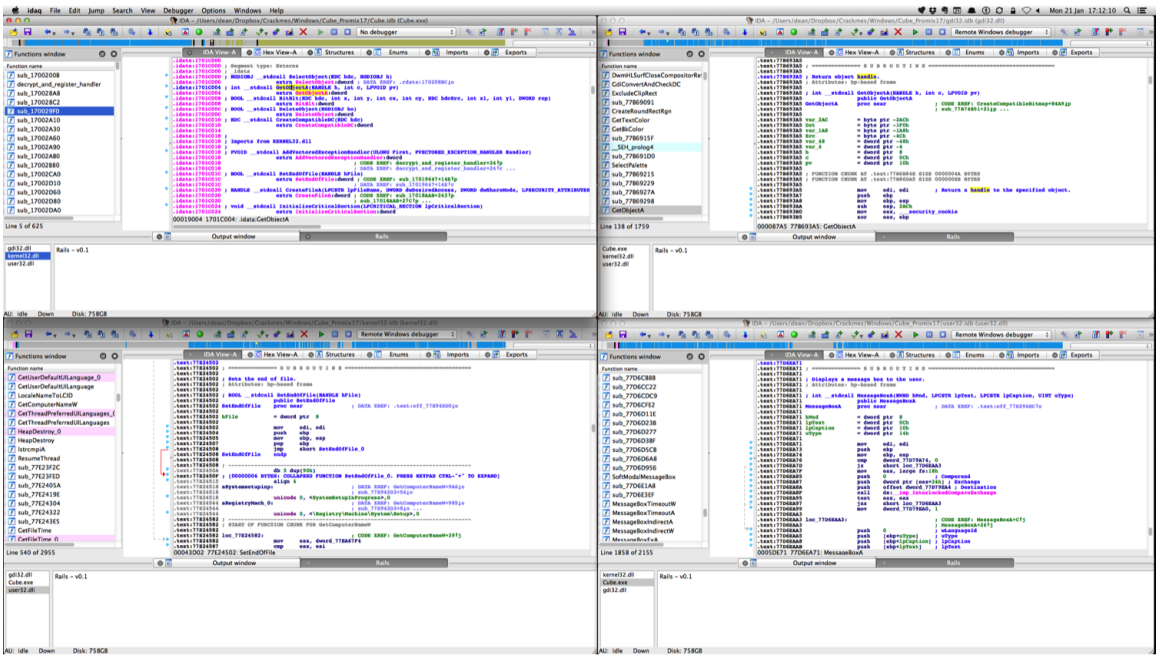

Figure 2.1: Rails being used to analyze an a crackme

As a result of the requirements solicited in [7] and discussed in Section 1.5, there was a clear need for functionality in IDA Pro that would ease the process of working with multiple binaries simultaneously.

Rails, Figure 2.1, is a plugin I developed for IDA Pro that facilitates communication between multiple instances. It allows for comments to be propagated between instances, eases navigation between instances, and significantly cuts down on the duplication of work when analyzing binaries that leveraged dynamic libraries. Rails

was submitted to the 2012 Hex Rays Plugin Contest and was given an honourable mention.

2.3 Design

The design of ICE leverages the *Model-View-Controller (MVC)* [47] design pattern in which information from any *Executable Entity*, a binary or intermediate language representing assembly code, is stored in an extensible data model and visualizations act as views of that data model describing the Executable Entity being analyzed. The primary reason the MVC pattern was selected is because it allows for multiple visualizations to be created using a single description of the Executable Entity in question—enabling the creation of new visualizations in a way that is more language-agnostic than previous approaches in this domain.

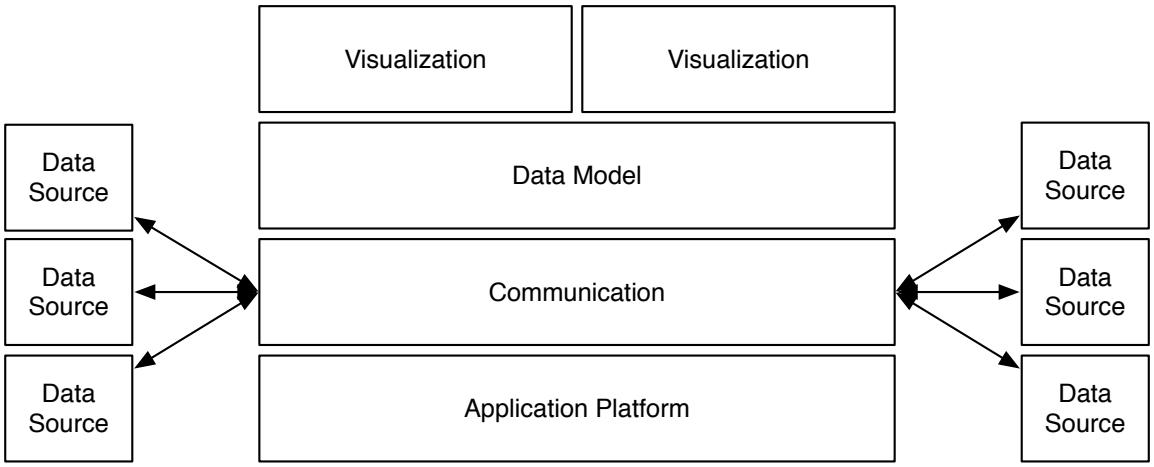

Figure 2.2: High-level design of ICE

A schematic representation of the design of ICE is shown in Figure 2.2. The lowest layer, the *Application Platform*, is responsible for providing all aspects of a graphical user interface and application. This layer includes functionality such as window management, delivering mouse and keyboard events, and a minimal environment for the creation of an application.

Above the Application Platform is the *Communication* layer. The Communication layer enables bi-directional communication between ICE and external applications. Within ICE the external applications are referred to as *Data Sources* and ICE is known as a *Data Sink*. Although this terminology implies that information only flows

from sources to the sink, ICE is capable of pushing changes—such as added comments and function name changes—to the sources.

Directly above the Communication layer is the *Data Model*; the "Model" in traditional MVC terms. The Data Model is a core component of ICE and forms the foundation upon which analyses are built, and data pertaining to low-level representations being analyzed is stored. The Data Model is encapsulated in a directed graph that models the structure of the Executable Entity under analysis. Within an Executable Entity function calls are used as the "edges" of this model since they are a ubiquitous mechanism to connect sections of code. For example, Figure 2.3 depicts the directed graph representation of a small program in which `main()` calls `foo()` and `bar()`, and `bar()` calls `print()`.

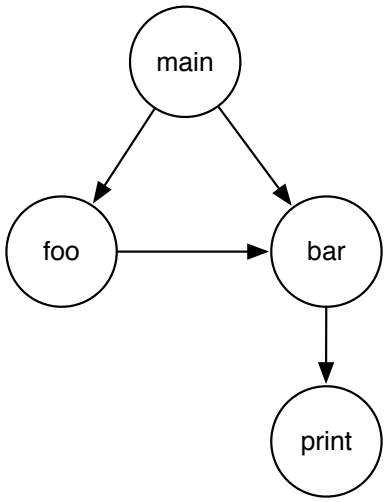

Figure 2.3: Graph representation of a program

Above the Data Model, ICE provides a mechanism for the development of visualizations that can easily be created by an analyst. Visualizations are the components that an analyst interacts with and are the viewport into the data model, or the "View" in MVC. The MVC paradigm is rounded out with the "Controller" being the logic of ICE that enables a user to switch between the various Views and interact with them to better understand what is being presented.

2.4 Implementation

Given the overall modular design of ICE outlined in Section 2.3, we carried out the prototype implementation by composing several existing technologies.

ICE was written in Java using the *Eclipse Rich Client Platform (RCP)* [25] as its foundation. The selection of this environment was guided in part by the prevalence of Java and the Eclipse RCP; however, it also allowed for a large amount of code reuse, specifically in the Tracks visualization (Section 2.4.3) and the Zest framework [65] for rendering graphs. The following subsections delve into the current communication, data model, and visualizations present in ICE.

2.4.1 Communication

As previously described in Section 2.3, ICE allows bi-directional communication between Data Sources and itself. Communication is carried out over a predetermined port on the loopback interface. Taking this approach restricts communication to the localhost which cuts down on network traffic and is beneficial in the context of malware analysis, since machines dedicated to this task are typically separated from all networks to promote security.

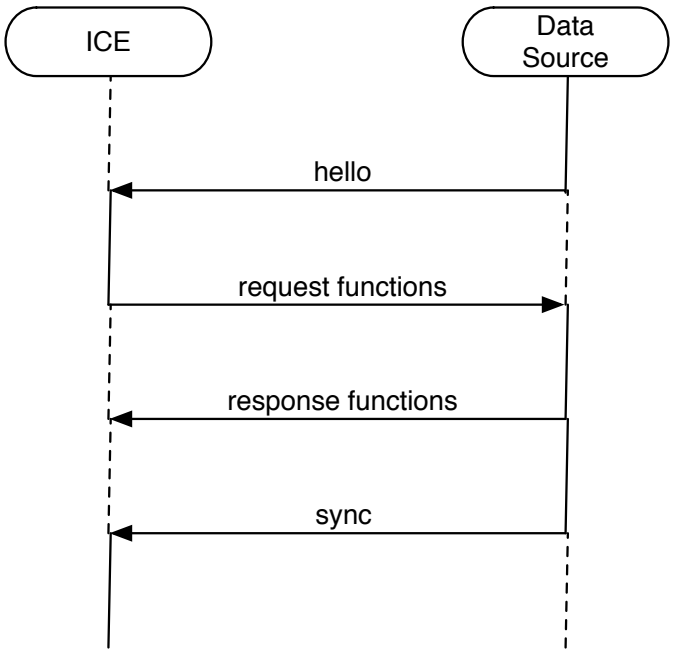

Figure 2.4: Message passing between ICE and a data source

The communication protocol used in ICE is built upon the JSON model [35] and is outlined in Figure 2.4. Communication begins by a Data Source sending a `hello` message to ICE. Upon receiving this message ICE creates a new entry in its Data Model that uniquely identifies the sender. A sample `hello` message is shown in

Listing 2.2.

Listing 2.2: Sample JSON message

```
{
        origin = ``program.exe'',
        instance_id = 1234,
        action = ``hello'',
        actionType = null,
        data = null
}
```

The fields presented in Listing 2.2 have all been chosen to allow for a flexible messaging protocol. The *origin* is the name assigned to the binary by the Data Source. For example, this could be the actual name of the binary or some other identifier such as a project name if Java byte-code was being analyzed. The next field, the *instance_id*, is a unique numeric identifier of the Data Source. Currently this is taken to be the process ID of the Data Source since that is guaranteed to be unique due to ICE being restricted to a single machine. The *action* describes what the message does—it can be thought of as the 'verb' in the message. The messaging protocol defines numerous *action* values including: `hello`, `request`, and `response`. Similarly, the *actionType* field can be thought of as the 'adverb' of the message since it describes the action of the message being sent. Finally, the *data* field is specific to the combination of *action* and *actionType* and can contain any valid JSON object.

Once the entry in the Data Model has been created ICE then requests information about the functions contained in the Executable Entity—a binary or intermediate language representing assembly code—under analysis. For each function ICE requests:

- Module name
- Function name
- Entry point
- Starting location
- Ending location
- Comment

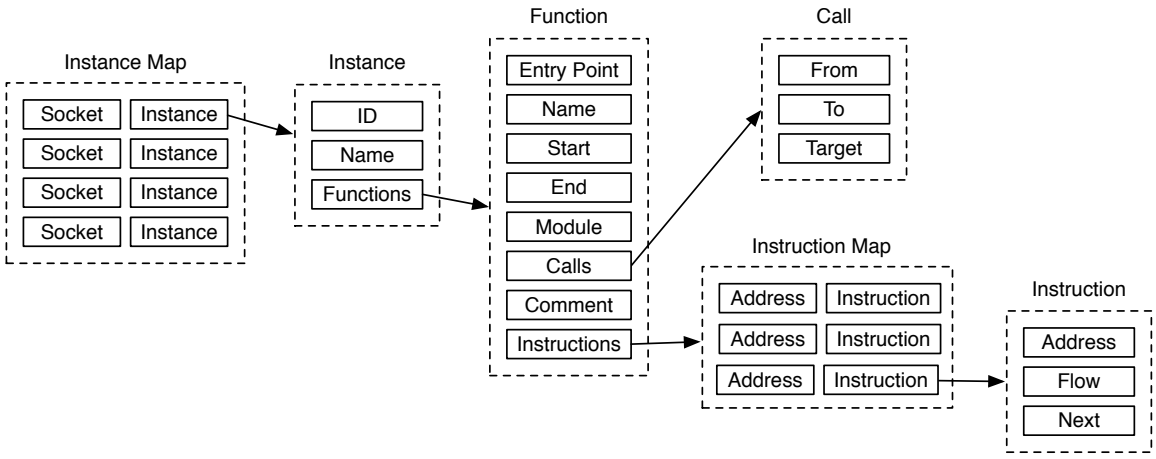

Figure 2.5: ICE Data Model containing relationships between modules, functions, and instructions.

Note that the starting and ending location may be either an address or a line number depending the Executable Entity being analyzed. Moreover, the *Entry point* is a boolean value indicating if the function is reachable from outside the Executable Entity and the function request triggers the Data Source to return similar information about all calls made within the function.

Upon sending all requested information, the Data Source sends a `sync` message causing ICE to commit all the data it has received, analyze the data for relationships between functions, and notify any visualizations currently open to update their view.

2.4.2 Data Model

The Data Model in ICE, depicted in Figure 2.5, is based on a directed graph and models the relationships between modules, functions, and instructions. Since ICE is able to support multiple Data Sources, the top-level of the model is an *Instance Map*. The Instance Map serves the purpose of mapping each *Instance* onto its corresponding communication socket.

For each connected Data Source there is an *Instance* object that describes it. The Instance object contains metadata such as the instance identifier and name, along with a hash table containing a mapping of locations to *Function* objects. Each Function object contains the following metadata:

- **Entry Point:** Boolean value indicating if the function is an entry point.
- **Name:** Name of the function.

- **Start:** Location the function starts at.
- **End:** Location the function ends at.
- **Module:** Name of the containing binary.
- **Comment:** Associated comment (if any).

In addition to this metadata, each Function object contains a list of *Call Sites*. These Call Sites contain pointers to their target Function object, creating a graph of functions.

Lastly, the Function object contains a mapping of locations to *Instructions*. Each Instruction object is comprised of the following attributes:

- **Address:** Location of this instruction.
- **Container:** Location of the function containing this instruction.
- **Flow Type:** The "flow" of the instruction (e.g. normal, jump, call).
- **Next:** List containing pointers to the next instruction(s).

Through this directed graph model of an Executable Entity it is possible to apply existing graph analysis algorithms to explore the Executable Entity at the function- or instruction-level as well as analyze the relationships between multiple Data Sources, potentially representing multiple Executable Entities. This also enables ICE to show correspondence between several levels of abstraction such as a high-level code base coupled with its binary.

2.4.3 Visualizations

The final major components of ICE are the visualizations. Visualizations are the primary user interface element, and allow an analyst to look inside a program. Currently ICE contains three visualizations: simple call graphs provided by *Cartographer*, sequence diagrams provided by *Tracks*, a *Control Flow Graph (CFG)*, and *Tours*. Due to the extensible design and implementation leveraging the Eclipse architecture, creating new visualizations is a straightforward process discussed further in Section 4.

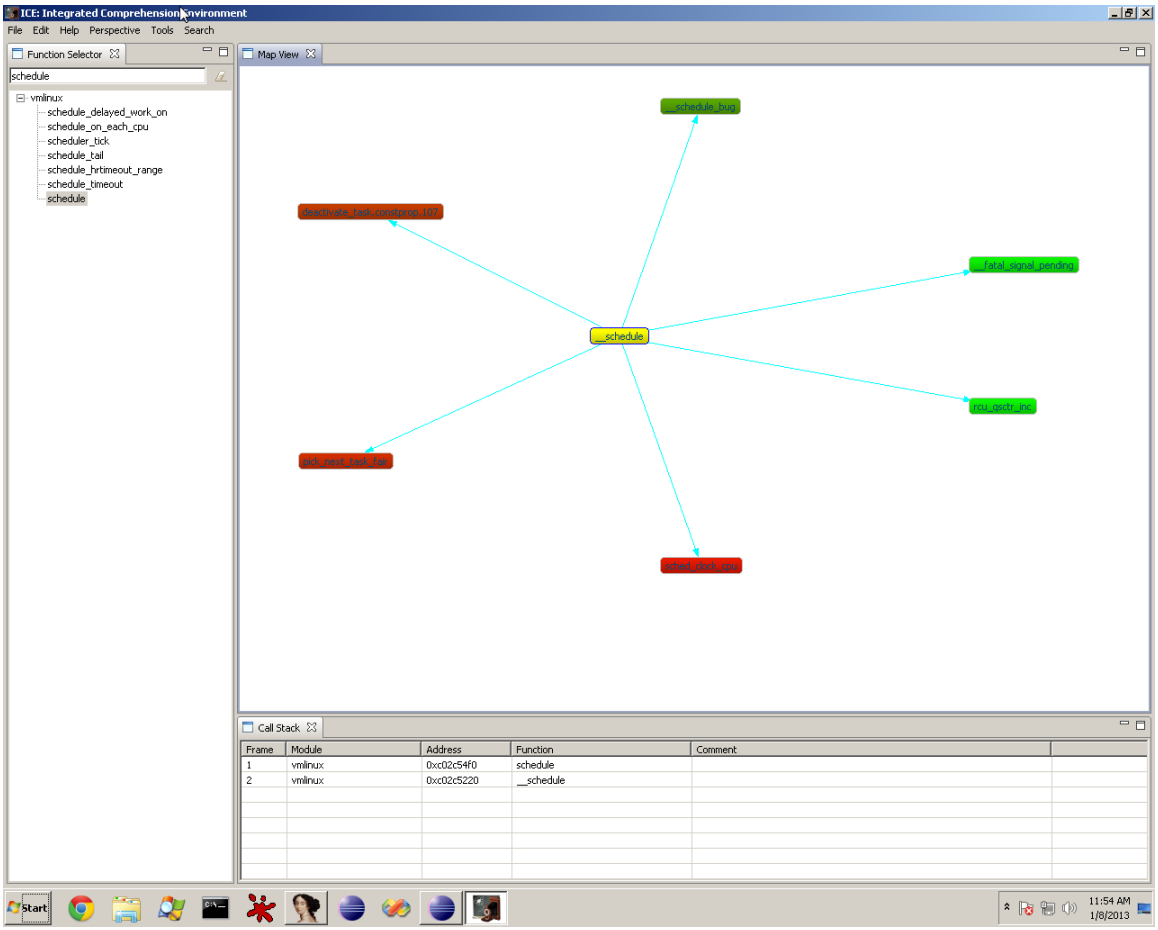

Figure 2.6: Screenshot of Cartographer

Cartographer

At the core of Cartographer, Figure 2.6, is a function call graph [49, 11], generated as ICE receives information about functions from its Data Sources. While the function call graph is a core component of Cartographer, there are two additional central aspects: interactivity, and navigation.

The key to the interactivity of Cartographer is that the call graph is not static, meaning that it accepts modifications by the analyst. Actions that allow the analyst to gain insight and manage the process of program comprehension include:

- Assigning names to functions
- Assigning comments to functions
- Re-positioning nodes in the graph

- Navigating to associated code in the Data Source

With the ability to assign names and comments to functions the analyst is able to better track portions of code that have been analyzed as well as assign meaningful names to functions. For example, a default function name in IDA Pro is of the form `sub_<address>` where `<address>` is the start address of the function. This name leaves much to be desired and a descriptive name such as `decryptCode` would provide the analyst a much clearer idea of what the function does. Similarly, with comments the analyst is able to better describe what the purpose of a function is and track functions that have been analyzed.

Additionally, the analyst is able to place the nodes in a visualization on the screen as desired, allowing for logical groupings—such as *these nodes have been analyzed*—and to manage clutter in complex functions. The colour of the nodes is also used to indicate the number of instructions in a function. Nodes that have a greener tint to them are shorter in terms of the number of instructions and nodes that have a redder tint to them are longer. The colour helps quickly identify functions that may be potentially more complicated.

With respect to navigation, Cartographer supports navigation within the call graph and navigation to the code. Navigation within the call graph is achieved by double-clicking a node which will cause the call graph of the selected function to be displayed. In addition to displaying the call graph of the current function, Cartographer also keeps track of calls made—by doubling-clicking a function—in a call stack. The call stack displays a sequential view of all the calls made by the analyst along with information such as the address, name, and comment associated with the corresponding function. The call stack also supports navigation. Finally, it is possible to further navigate to the code in a function from Cartographer—in this case the function will be opened up in the containing Data Source.

Tracks

In previous work we developed Tracks [5] (Figure 2.7), a visualization tool that displays function calls within a program as a sequence diagram for both static and dynamic control flow. Through the sequence diagram an analyst is able to gain insight into the functions called as well as the order in which the calls are made. Tracks additionally shows calls to functions in external libraries as well as provides loop detection. Actions from the user are also supported in the connected Data Source, such

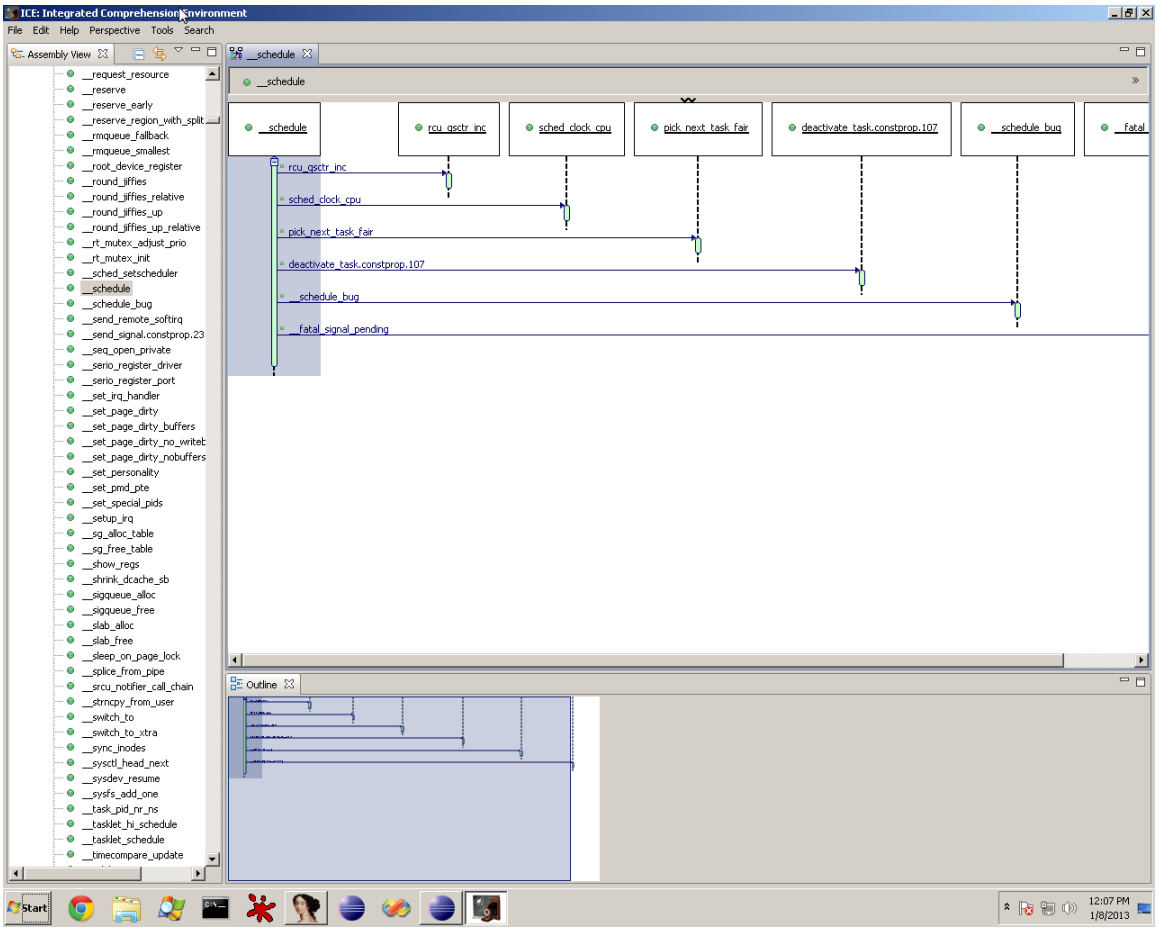

Figure 2.7: Screenshot of Tracks

as navigation to the code (either the function or specific call), setting breakpoints, and syncing renamed functions. Tracks was refactored to adhere to the design discussed in the previous Section, which allowed us an initial analysis of the plugin architecture that is used in ICE.

Tracks was built on top of Diver [10] to support extremely large traces so provides features such as hiding/collapsing call trees and package or module structures, setting new roots of diagrams, navigable thumbnail outline view, and saving the state of the diagram. Previous work has also investigated the use of comment threads within the sequence diagram itself [6].

Control Flow Graph (CFG)

A Control Flow Graph (CFG) makes it possible to become better acquainted with the inner workings of a function by identifying key structures such as loops and

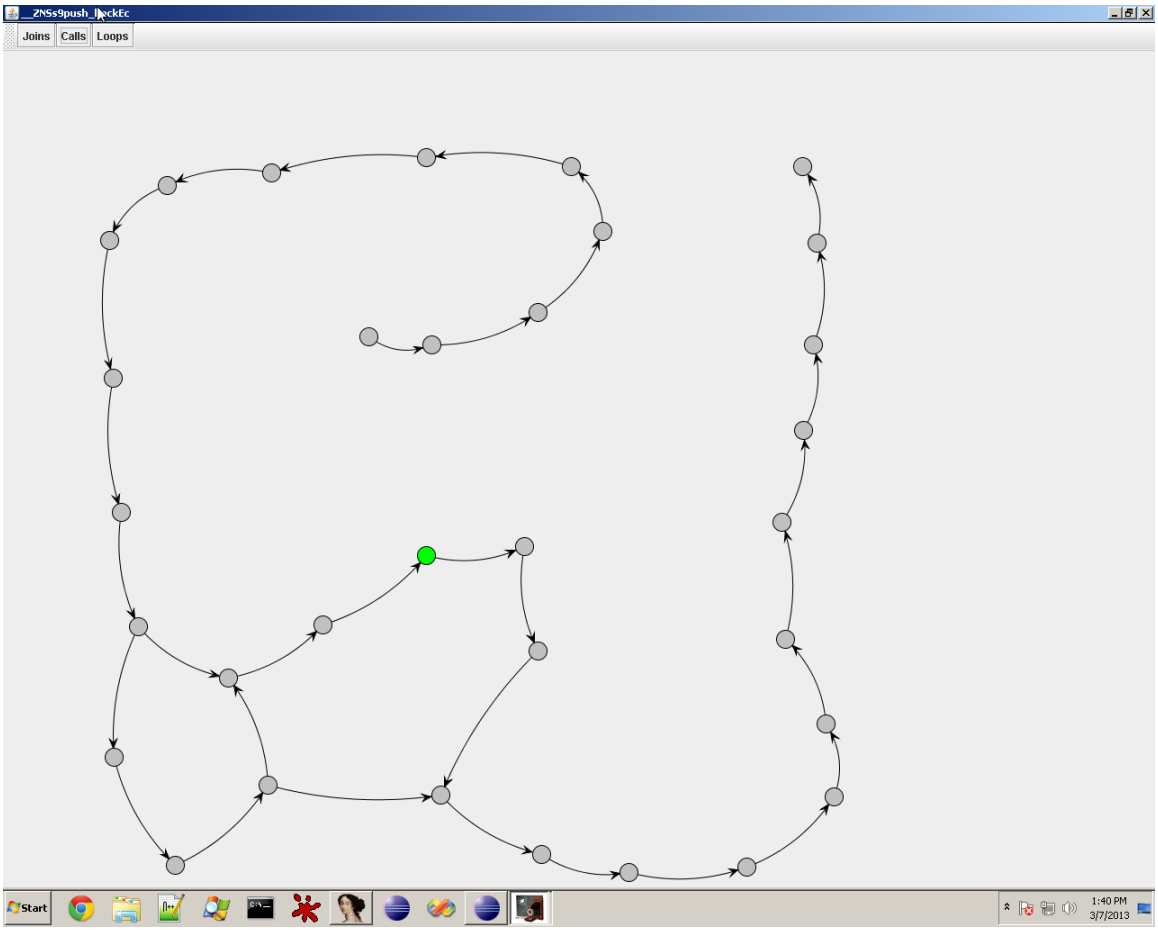

Figure 2.8: Screenshot of Control Flow Graph

branches. As with Cartographer and Tracks, the CFG visualization—Figure 2.9—is also interactive and provides filters to help pinpoint how instructions are related.

With respect to interactivity, the CFG visualization supports zooming in and out, panning, and rotating the nodes of the graph. This is particularly important in this domain, where the number of lines of code in a function may have exploded several orders of magnitude relative to its high-level representation. In addition, individual nodes can be selected and moved to arbitrary locations, once again aiding in clutter management and comprehension.

The novel aspect of the CFG related specifically to comprehension is the ability to select from a set of filters. Each filter highlights the associated set of nodes making them visually discernible and easy to spot relative to the other nodes shown. The CFG in ICE provides filters for: Calls, Joins, and Loops. The Call filter simply highlights all `call` instructions and can be used to correlate where a call is located

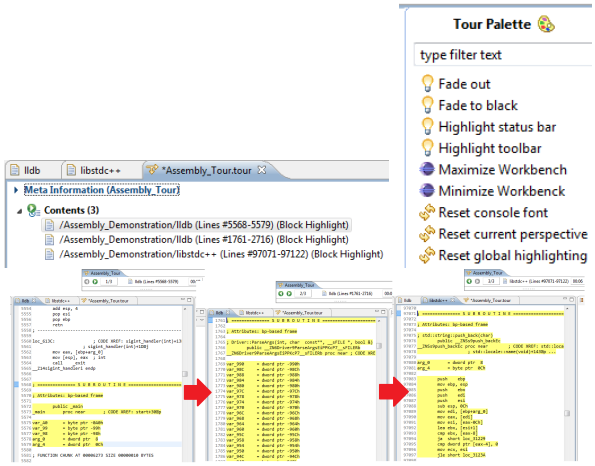

Figure 2.9: Screenshot of Tours

within a function without the need to analyze the assembly code. The Joins filter highlights all nodes that have an in- or out-degree greater than one. These nodes represent locations in the function where high-level control flow constructs such as if-then-else, try-catch, switch, and other related statements are found. By identifying these nodes it can be seen if certain instructions have an abnormal number of incident edges aiding in the identification of "interesting" locations in the function. Finally, Loop detection is based on the Tarjan strongly connected component [57] algorithm.

Tours & TagSEA

Finally, given the challenges of scale and multiple levels of abstraction, in order to better support the transferal of knowledge between developers in these domains, ICE further leverages an existing Eclipse based plugin called *Tours* [12, 55]. Tours was developed in previous work at the University of Victoria in cooperation with IBM in order to provide programmers a lightweight method of developing walkthroughs of source code [12].

The key benefit of this plugin is that it does not require analysts to jump between environments to understand correspondence between high- and low-level code. Tours is used essentially as a means of providing comprehensive documentation, providing a predetermined path around the code, but still allowing an analyst to explore areas of interest on their own. The tool is meant to help with the transfer of knowledge between programmers. In ICE, Tours can currently be used on any representation of code. To create a tour, the user selects lines of code which a presentation will flow

through, demonstrating significance between the segments. The tool comes with a number of presentation inspired features including highlighting and dimming of the workspace. The plug-in uses an XML representation of the line number and file name to create a tour point.

2.5 Summary

In this chapter we explored the design and implementation details of ICE as well as the guiding principles. The fundamental data structure, a directed graph, was introduced along with how it can help support the remainder of ICE. The model behind ICE was discussed and, finally, three visualizations—Cartographer, Tracks, and a Control Flow Graph—were introduced.

Chapter 3

Case Studies

This chapter presents three case studies that investigate different aspects of program comprehension. The first two case studies analyze a binary with notable characteristics using ICE, IDA Pro, and Hopper. Hopper is a disassembler recently released for Mac OS X and has quickly gained popularly within the Mac OS X reverse engineering community. The final case study investigates the ability for ICE to be integrated into existing Data Sources.

3.1 Case Study: Dynamic Linker

This case study investigates the ability of an analyst to comprehend an algorithm that has been implemented. This technique could be used to verify that an implementation is accurate or to identify weaknesses in an implementation. For this case study the *dynamic linker (dyld)* that is used in Mac OS X and on iOS devices is analyzed. The implementation of dyld being analyzed is from iOS and is available from Apple's open source repositories.

3.1.1 Overview of dyld

Before analyzing the implementation of dyld used in iOS it is beneficial to first define the scope of the case study and briefly discuss how an executable is organized in iOS.

On iOS executables use the Mach-O architecture. Mach-O binaries consist of a header followed by the required number of segments for the program being executed. Among other information, the header contains a list of *load commands*. These load

commands are what dyld uses to properly load the segments in the executable and handle other details that may be present in the executable.

In this case study we want to analyze the functionality that is executed up until the point where the load commands have been parsed. To do this our entry point into dyld will be the `dlopen()` function which has been selected due to its presence across multiple UNIX-based systems and its intention as a way to load a dynamic library.

3.1.2 Analysis with ICE

Using ICE an analysis of dyld was carried out with the goal of understanding the process used to load an executable up to the point where load commands have been parsed. As a first step in this analysis the executable for dyld was loaded in IDA Pro and then ICE was started. With ICE open, `dlopen()` was found in Cartographer as seen in Figure 3.1. A cursory look at `dlopen()` results in a lot of information to digest and no clear method of approaching it. To give the analysis some direction, we switch to Tracks and investigate `dlopen()` from that perspective.

Figure 3.2 is a screenshot of `dlopen()` viewed in Tracks. Since Tracks displays function calls ordered by the address of the call [1] a best guess of which function to move to next is `dyld::load(LoadContext *)` [2].

Within `dyld::load(LoadContext *)` 3.3 it is seen that a function exists named `dyld::loadPhase0()` and it is selected as the best candidate to move forward. This selection was made because the function is called twice from `dyld::load()` as seen by the two lines connecting the nodes and the name hints at it being the path used to load an executable—our goal for this case study.

Continuing with `dyld::loadPhase0()` 3.4 we see that there is a call to a function named `dyld::loadPhase1()`. The naming convention being used hints that loading an executable occurs in multiple stages.

Investigating `dyld::loadPhase1()` we discover that both `dyld::loadPhase2()` and `dyld::loadPhase3()` are called. Both Cartographer and Tracks are unable to help decide which path to take so we investigate both of them to discover that they both call `dyld::loadPhase4()`. This likely means that either phase two or

[1] The order of function calls in Tracks is not necessarily the same as the order that the calls may occur during execution.

[2] The function names shown are as IDA Pro has parsed them and include the namespace-mangling used in C++

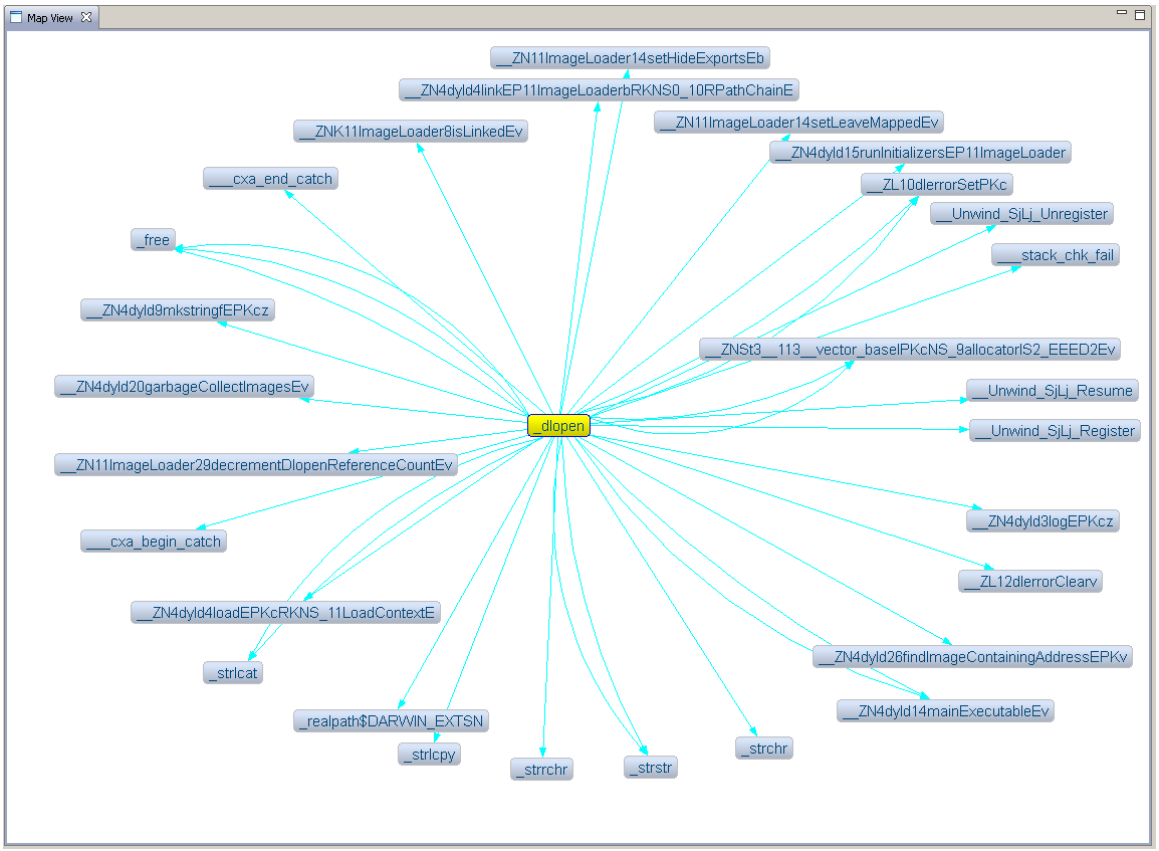

Figure 3.1: dlopen() as seen through Cartographer

phase three of the loading process are a fallback of some kind or used to handle a special case. From `dyld::loadPhase4()` we are taken into `dyld::loadPhase5()` and from there into `dyld::loadPhase6()`. In `dyld::loadPhase6()` 3.5 we do not find any references to further phases in the loading process but do find a call to `ImageLoaderMachO::instantiateFromFile()`.

Inside `ImageLoaderMachO::instantiateFromFile()` 3.6 there are calls to both "Classic" and "Compressed" variations of the loader. Without knowing the difference between these two variations we arbitrarily select the Classic variation. Following this path (Figure 3.7) we discover a call to `ImageLoaderMachO::parseLoadCmds()` which sounds like a match for the goal of this case study.

Using Cartographer (Figure 3.8) and Tracks to investigate `ImageLoaderMachO::parseLoadCmds(` does not give much insight into the implementation of the function. However, by focusing in on a control flow graph of the function we can gain some key insights before having to go to the level of reading and analyzing the assembly code directly.

ICE provides the ability to view, and interact with, a Control Flow Graph (CFG)

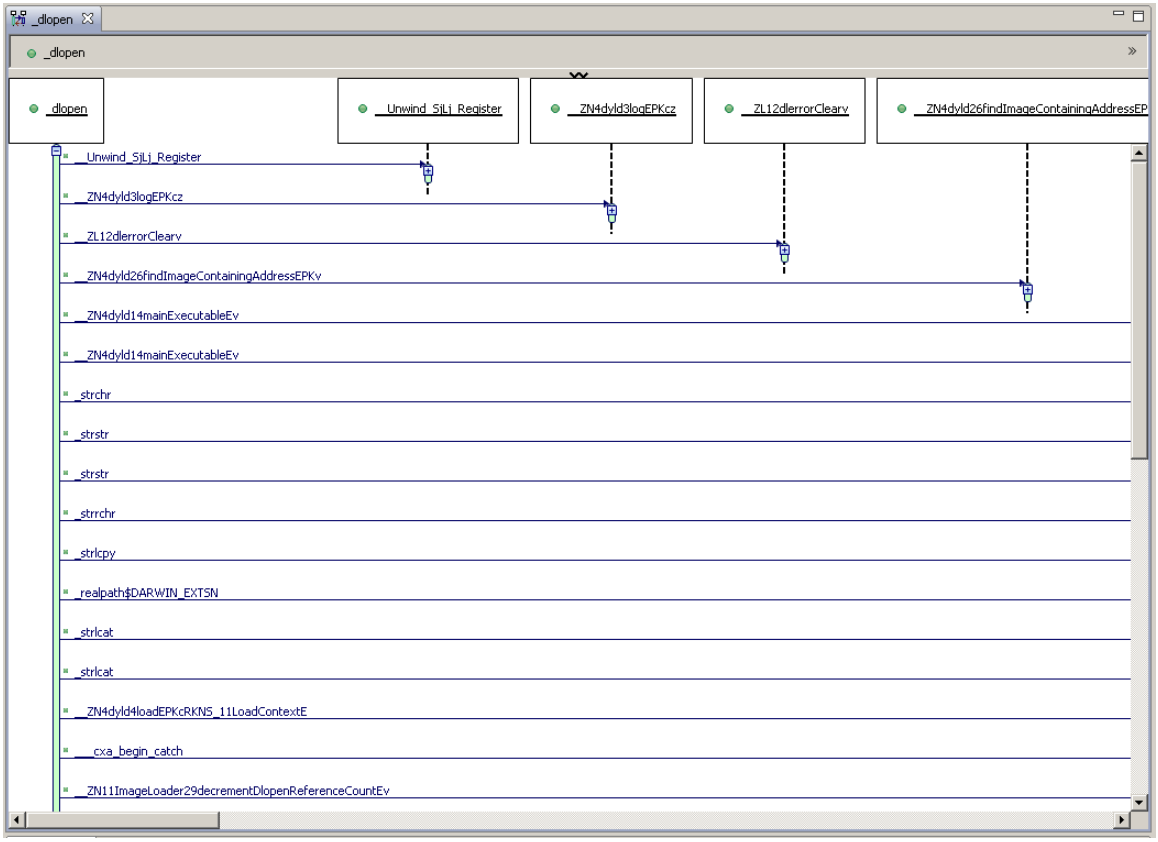

Figure 3.2: dlopen() as seen through Tracks

where each node represents an instruction in the function being examined. The CFG provides three filters to help deal with a potential overload of information: Calls, Joins, and Loops. Each filter works by highlighting the nodes so that they stand out among the rest. For our analysis of `ImageLoaderMachO::parseLoadCmds()` the Calls filter does little to help further our understanding due to the small number of calls made.

Figure 3.9 shows the CFG of `ImageLoaderMachO::parseLoadCmds()` with the Joins filter enabled. Each highlighted node identifies an instruction that has one or more incident edges. In Figure 3.9 the nodes labelled 1 and 2 indicate points in `ImageLoaderMachO::parseLoadCmds()` where a switch-like statement appears to converge. The reason this is not conclusive is that the labelled nodes could be points where other statements, such as a loop, converges. The switch statement is an educated guess based on the high number of incoming edges and our previous knowledge of the load commands used in Mach-O files.

If we now switch the CFG to highlight loops we are shown the graph in Figure 3.10.

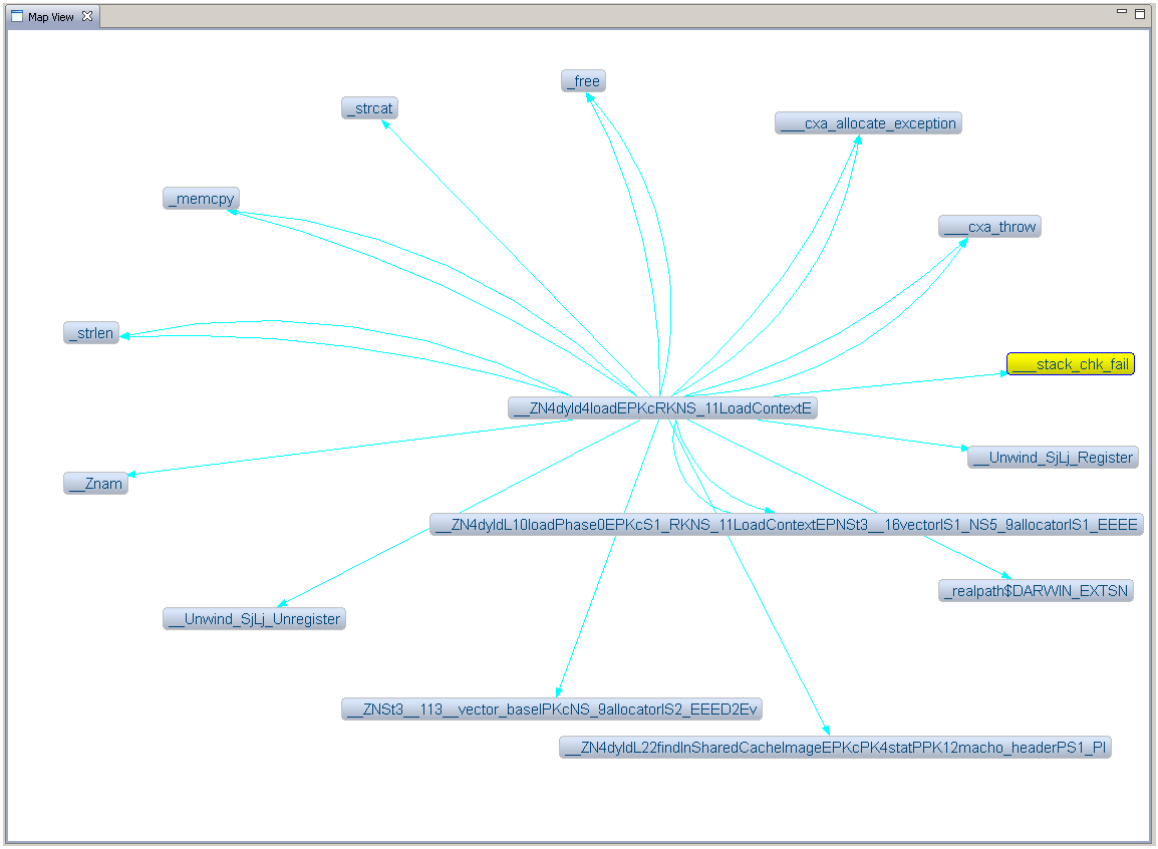

Figure 3.3: dyld::load() as seen through Cartographer

In this figure the node labelled 1 appears to, once again, be a convergence of multiple code paths. Similarly, the node labelled 2 appears to be the entry point of a loop where as the node labelled 3 appears to be some kind of a boolean check point.

From our analysis using ICE we have found a potential path from `dlopen()` to the code that parses Mach-O load commands. In the function for load command parsing we have identified numerous instructions that would be acceptable candidates for an analysis of the assembly code.

3.1.3 Analysis with IDA Pro

Analyzing the code executed by `dlopen()` to the point that the load commands are parsed with IDA Pro is now shown.

The first step taken during this analysis is to navigate in IDA Pro to the `dlopen()` function. Once here we view a graph of the references from `dlopen()`, Figure 3.11. As seen in the figure, this graph is exceedingly complicated. The complication largely

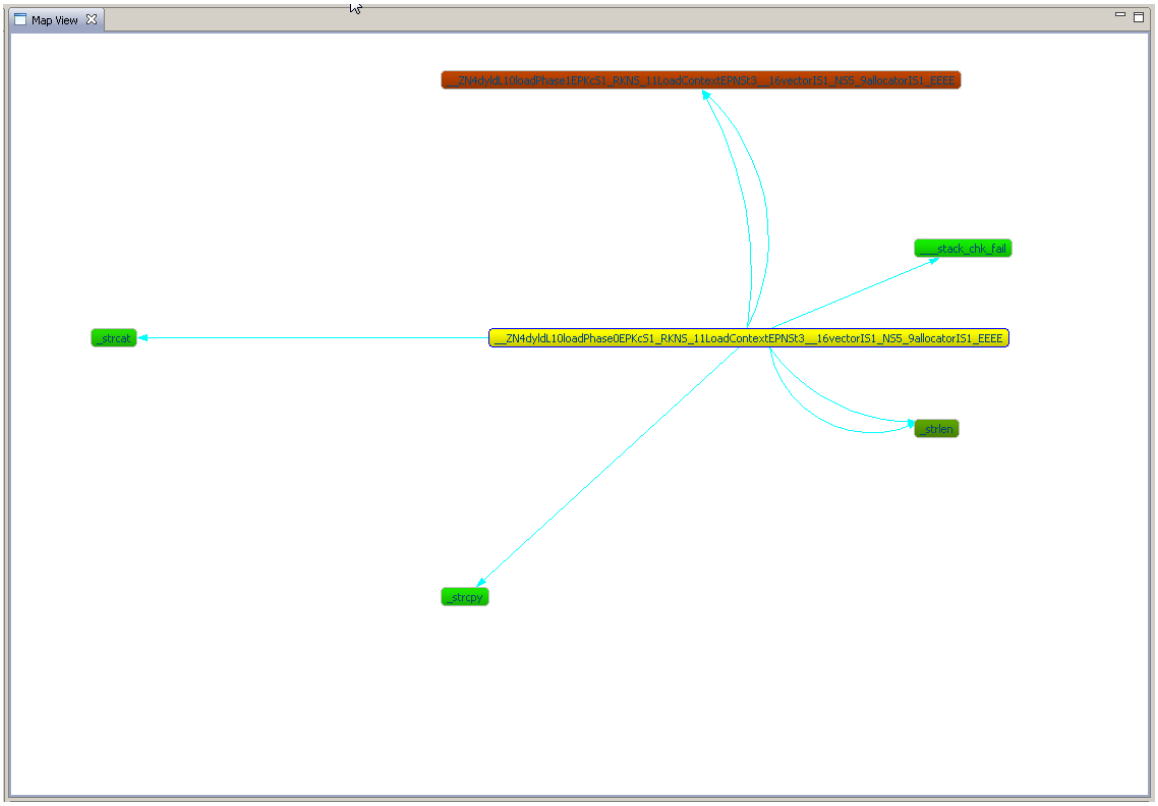

Figure 3.4: dyld::loadPhase0() as seen through Cartographer

arises from the fact that graph includes sub-references, that is references made by references from `dlopen()`, and the graph does not take advantage of the available screen real estate. The graph does support zooming and panning which helps mitigate some of the complexity; but it is not possible to interact with nodes in any way. To try and make sense of this graph a list of references from `dlopen()` was requested in IDA Pro (Figure 3.12), unfortunately IDA Pro informs us that there are no references to be displayed.

A recent update to IDA Pro unveiled a new graph called the *Proximity View*. The Proximity View displays all references (data and calls) to and from a selected function. The graph supports panning and zooming as well as the interactivity of nodes. The next step in our analysis of `dlopen()` was to view it in the Proximity View (Figure 3.13). To make Figure 3.13 more readable the number of child references was limited to 1, parents and data references are not shown, and the layout was set to radial rather than the default tree-like layout.

From the information shown in the Proximity View, we decided to follow the call

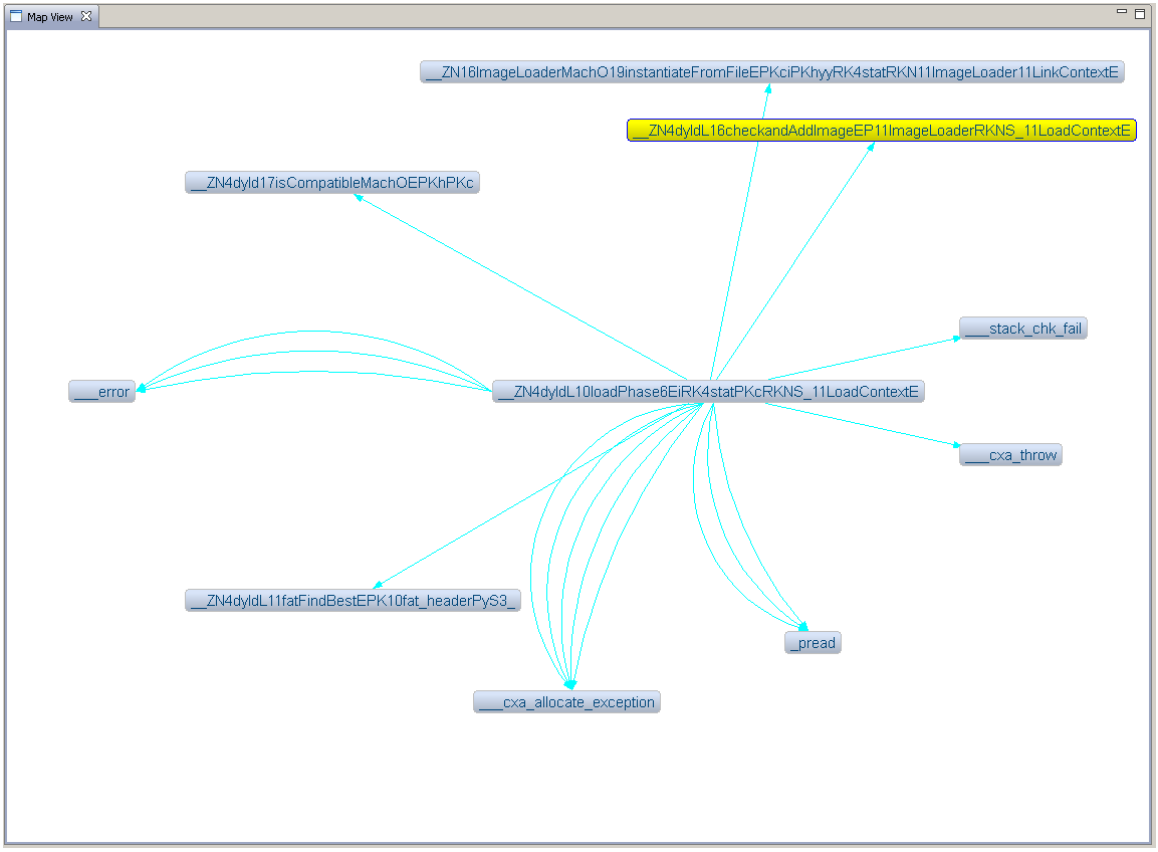

Figure 3.5: dyld::loadPhase6() as seen through Cartographer

to `dyld::load()` as was done when analyzing with ICE. Using the Proximity View as the primary method of discovering code paths the same series of function calls was traversed as was done with ICE. This path eventually yielded a call do the function of interest for this case study which is `ImageLoaderMachO::parseLoadCmds()`.

The Proximity View of `ImageLoaderMachO::parseLoadCmds()` is shown in Figure 3.14. Like Figure 3.13, the Proximity View has references to data and parents disabled as well as the number child levels to display set to 1. The default layout was used for this graph because the graph is somewhat simple. From Figure 3.14 it is not clear what the `ImageLoaderMachO::parseLoadCmds()` function does so a closer look is necessary.

Figure 3.15 is the control flow graph of the `ImageLoaderMachO::parseLoadCmds()` function. The first aspect of this graph that stands out is that the nodes represent basic blocks rather than individual instructions. The usage of basic blocks does cut down on the number of nodes displayed; however, when zoomed in the blocks display

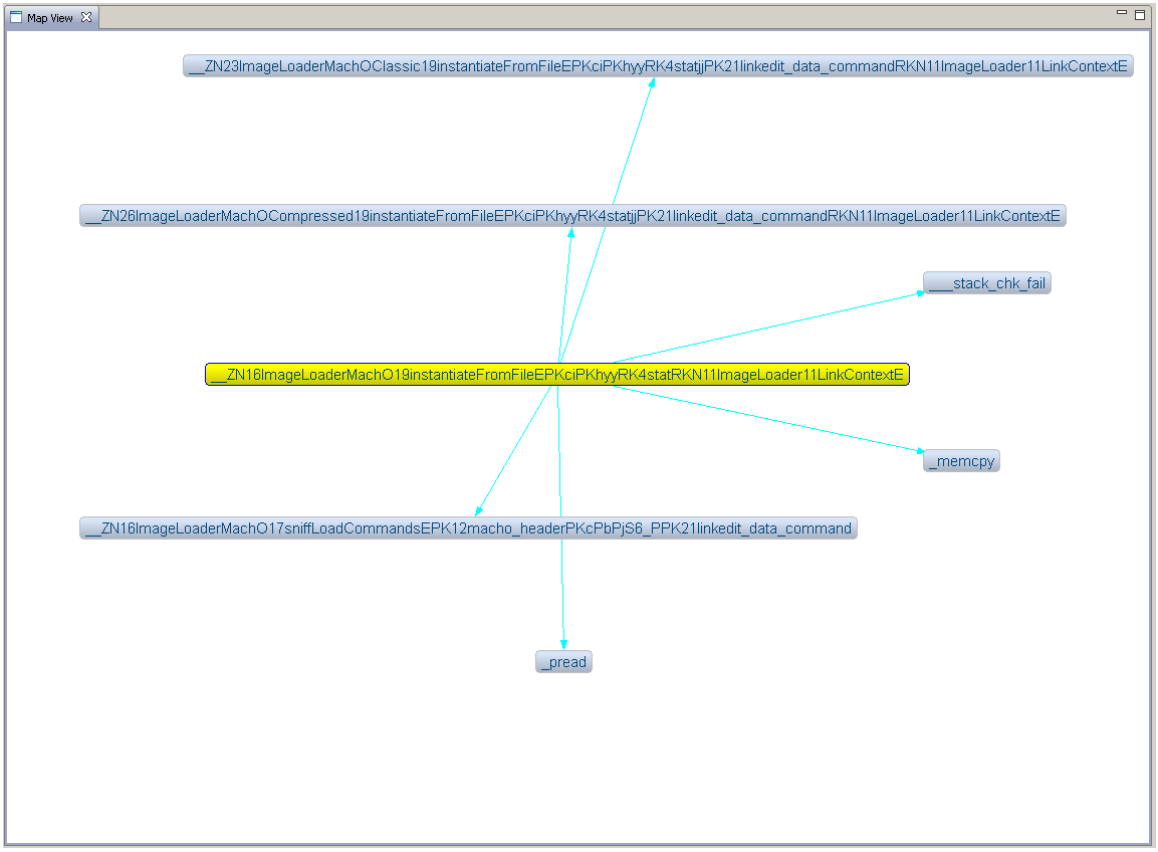

Figure 3.6: ImageLoaderMachO::instantiateFromFile() as seen through Cartographer

all the instructions contained the basic block so the nodes end up consuming a large amount of space. The second aspect of this graph that stands out is that it is difficult to identify the loops found in the function. To discover the loops it is necessary to (1) understand the flow of the code and (2) trace the jumps through the basic blocks.

The end result of this analysis was that ICE was able to display similar information in a manner that is a easier to understand due to the lack of assembly code being shown. It is also concerning that the graph of references from `dlopen()` displays a different set of information than the list of references from `dlopen()`.

3.1.4 Analysis with Hopper

Completing the analysis one last time with Hopper we begin by navigating to the `dlopen()` function. Unfortunately Hopper does not provide a way to view a function call graph so it is necessary to search (manually or via a script) through the assembly code looking for calls. Doing this search through `dlopen()` we find a call

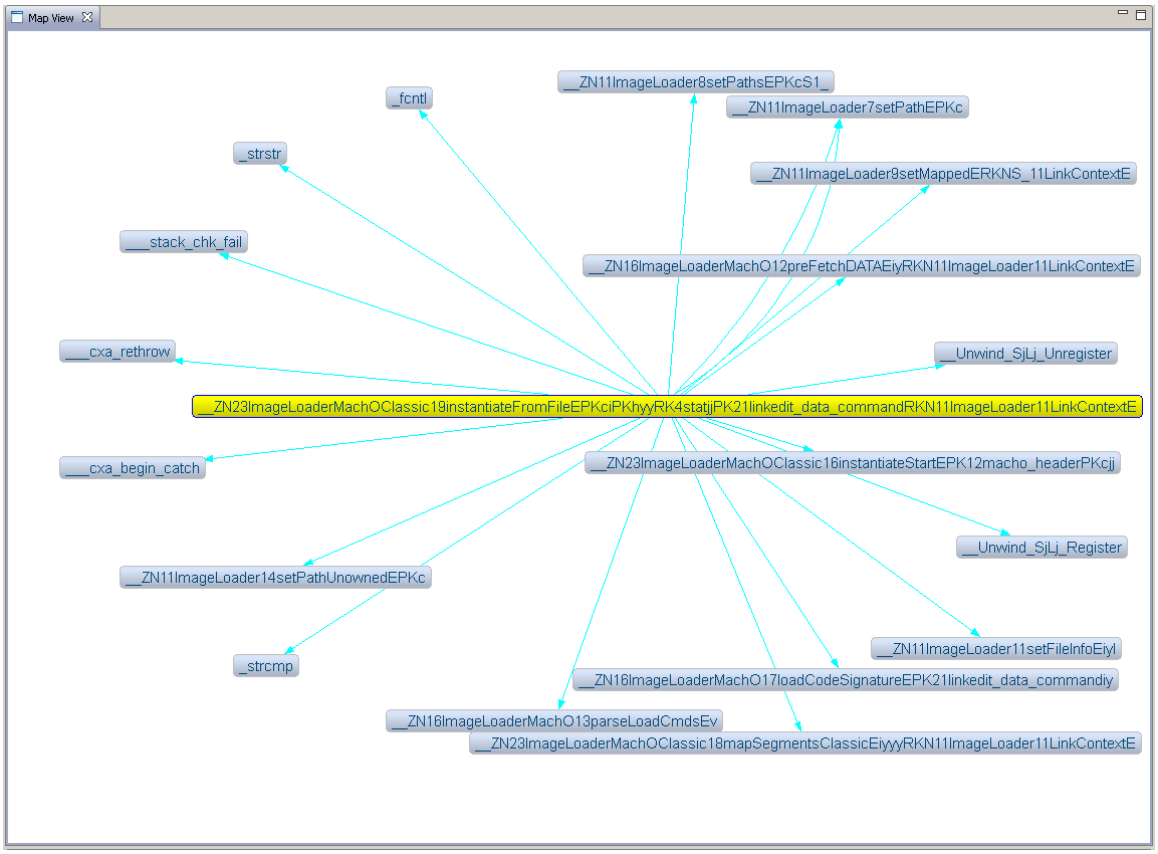

Figure 3.7: ImageLoaderMachOClassic::instantiateFromFile() as seen through Cartographer

to `dyld::load()` 3.16 after reading through 192 instructions. Furthermore, without the availability of a function call graph it is not known if any other relevant function calls occur after this point so `dlopen()` must be searched in its entirety.

Having performed the analysis with both ICE and IDA Pro we skip to the analysis of `ImageLoaderMachO::parseLoadCmds()` to leverage previous knowledge and limit the amount of manual searching required. As with `dlopen()`, our analysis of `ImageLoaderMachO::parseLoadCmds()` begins by first searching through the assembly to identify any function calls of interest. Like the analyses carried with ICE and IDA Pro this search yields no new information.

Switching our attention from the function at a high-level we generate a control flow graph with Hopper. Investigating the control flow graph, it is difficult to identify control flow structures and gain insight into the implementation of this function. This difficulty arises from the size, complexity, and amount of information shown on the graph.

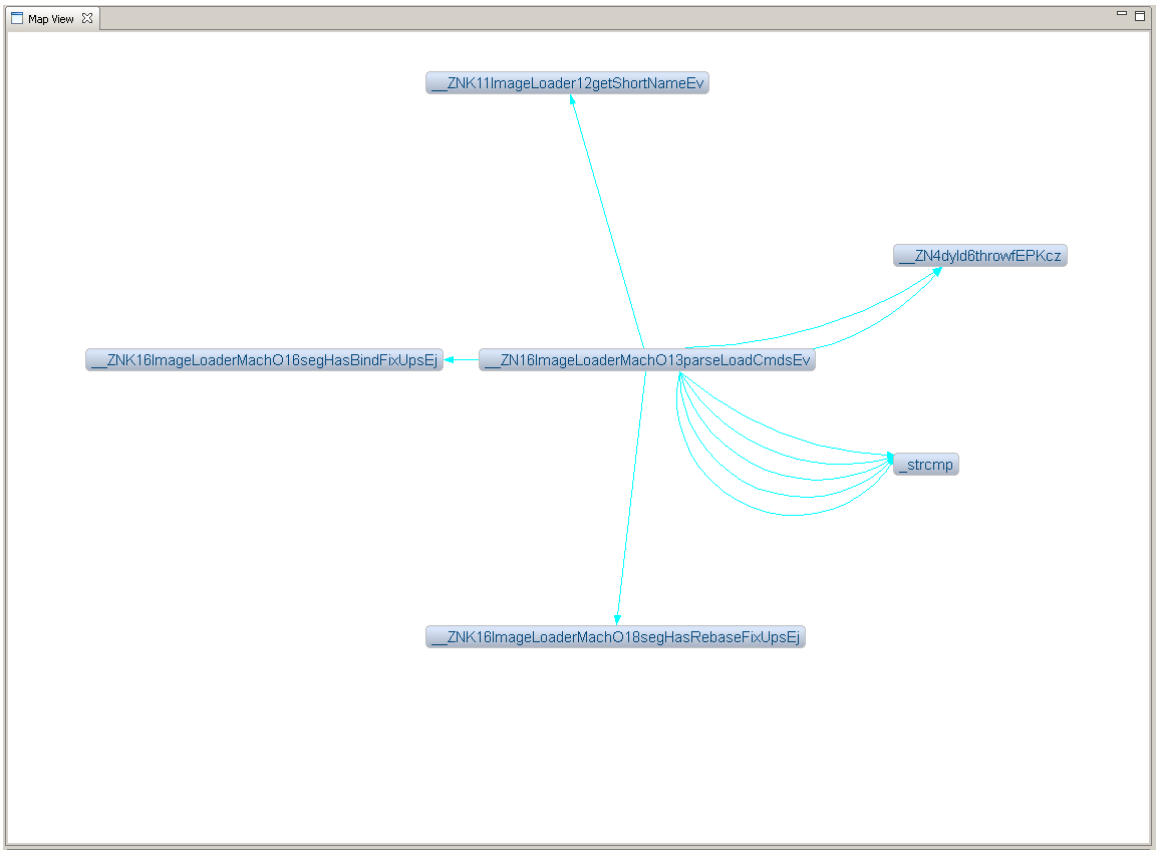

Figure 3.8: parseLoadCmds() in Cartographer

Through out the analysis of the dynamic linker with Hopper it was necessary to deal directly with the disassembly and it was not possible to leverage any high-level abstractions to aid the task of understanding how calling `dlopen()` leads to load commands being parsed.

3.1.5 Evaluation: Source Code

For this case study the authoritative source is the source code itself [3]. As with each analysis we begin by investigating the function `dlopen()`.

Table 3.1 compares the number of functions identified by ICE as seen in Figure 3.1 to the source code and IDA Pro. It is seen that ICE identifies 34 function calls where as there are 30 identified by IDA Pro and 40 calls made in the source code.

Since ICE uses information from IDA Pro it is important to note that the additional calls identified by ICE are a result of a function being called numerous times.

[3]Source code available at http://opensource.apple.com/source/dyld/dyld-210.2.3/

Figure 3.9: CFG of parseLoadCmds() with the Joins filter

For example, the call to `dyld::mainExecutable()` is made twice in the source code which is reflected in ICE. However, the graph used in IDA Pro—Figure 3.13—only displays connections for function calls made, not a connection *per* function call made.

The path from `dlopen()` to `ImageLoaderMachO::parseLoadCmds()` is as seen in both ICE and IDA Pro. Aside from identifying functions behind pointers and inline functions, ICE was able to assist in determining the correct path taken by code from `dlopen()` to `ImageLoaderMachO::parseLoad()`.

From this case study we have learned that ICE is capable of working with an ARM-based codebase and that it is capable of displaying in a manner that facilities an analyst to discover the correct path taken by the code.

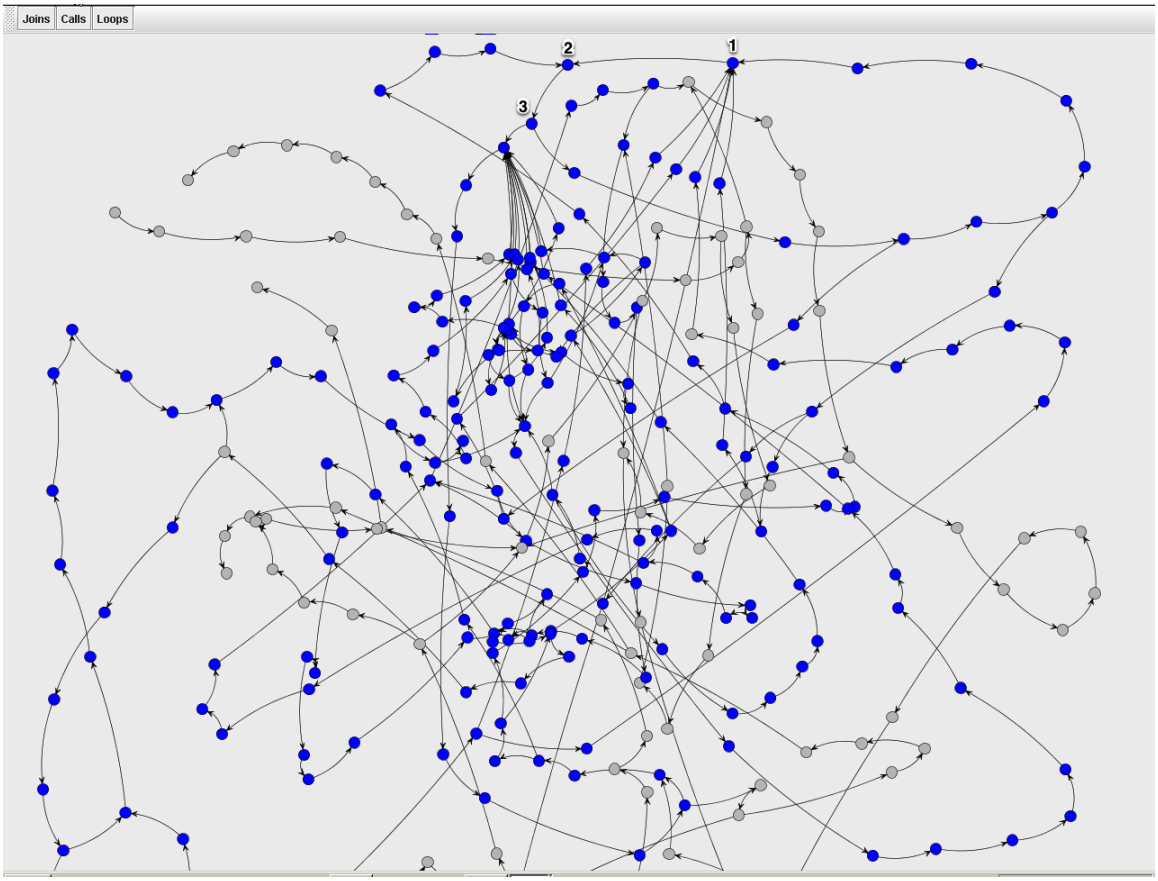

Figure 3.10: CFG of parseLoadCmds() with the Loops filter

3.2 Case Study: Malware

The second case study looks at the ability to identify actions in malicious software. The study looks at the infection mechanism and system changes within a malware sample from the textbook *Practical Malware Analysis* [51].

3.2.1 Overview of Sample

The sample used in this case study is from Chapter 11 Lab 1 of the book *Practical Malware Analysis* [51]. The sample was chosen based on the criteria that it is not obfuscated and does not make use of self-modifying code. As a result of these criteria it is possible to analyze the sample without having to first reconstruct import address table or decrypt the code. The goal of this sample is malicious and it can still be extremely destructive to the infected system.

Analysis of this sample was carried out using Windows XP Service Pack 3 running

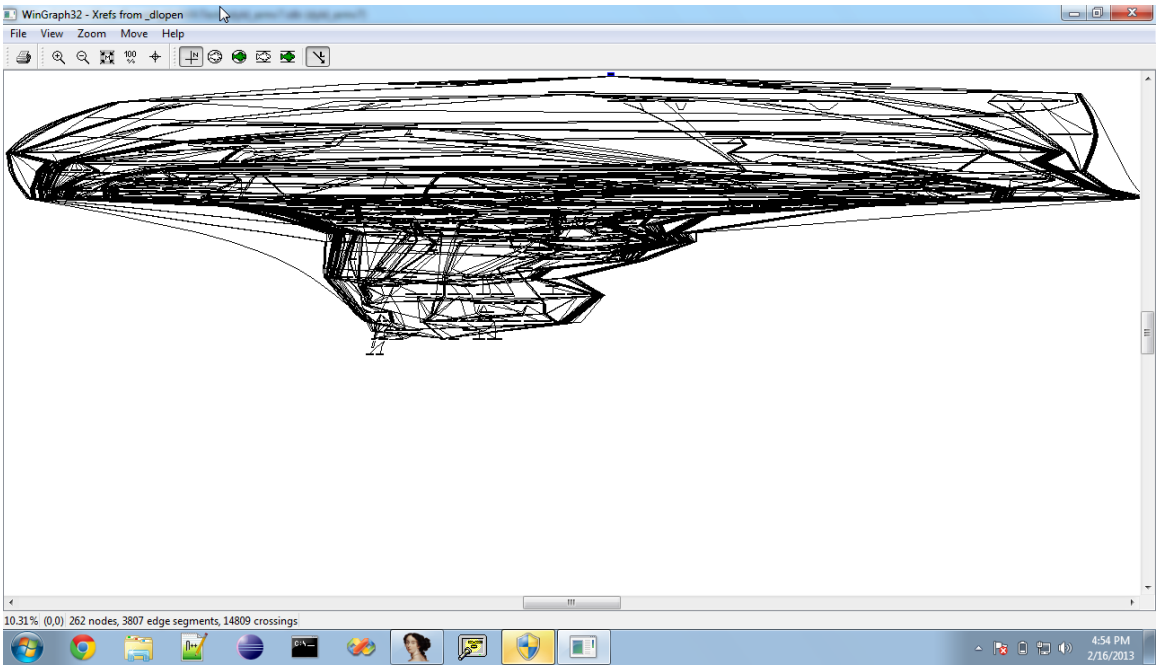

Figure 3.11: Graph of references from dlopen() in IDA Pro

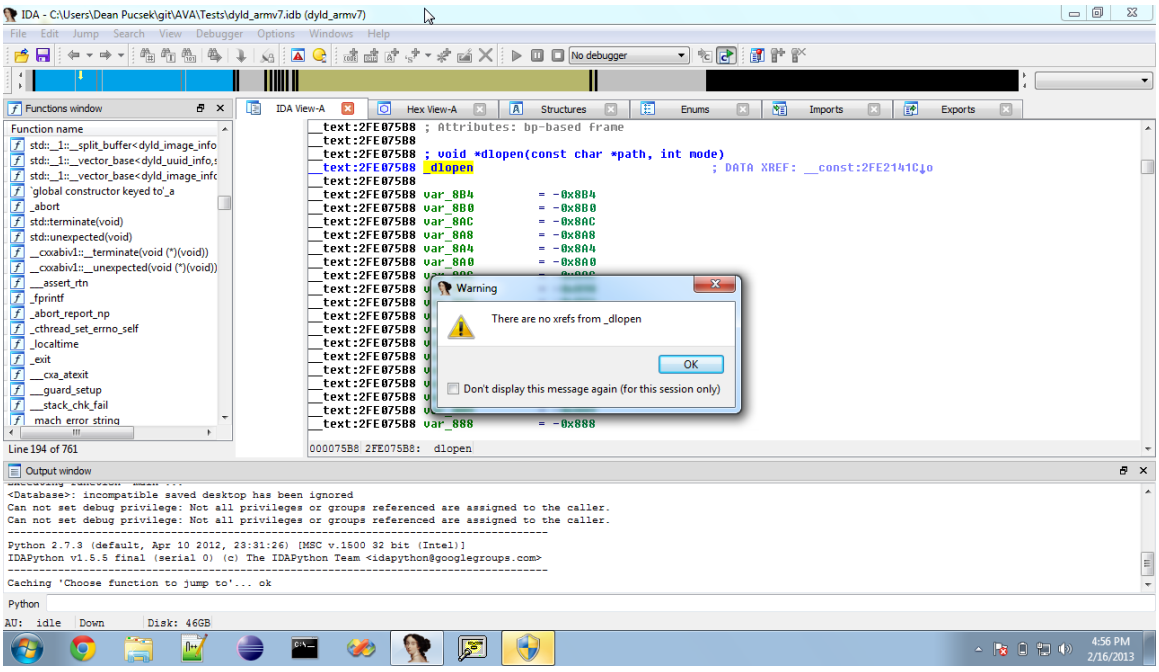

Figure 3.12: List of references from dlopen() in IDA Pro

inside a virtual machine. Before delving into the sample with and without ICE an initial analysis was performed as this is the first step when performing malware

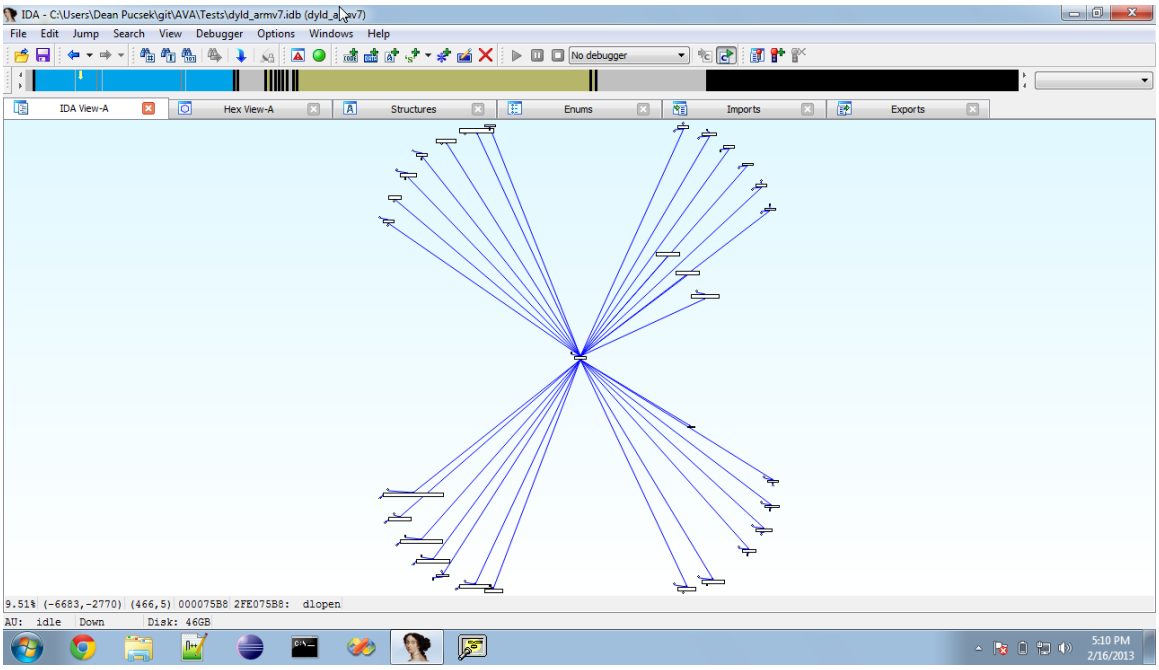

Figure 3.13: Proximity View of dlopen() in IDA Pro

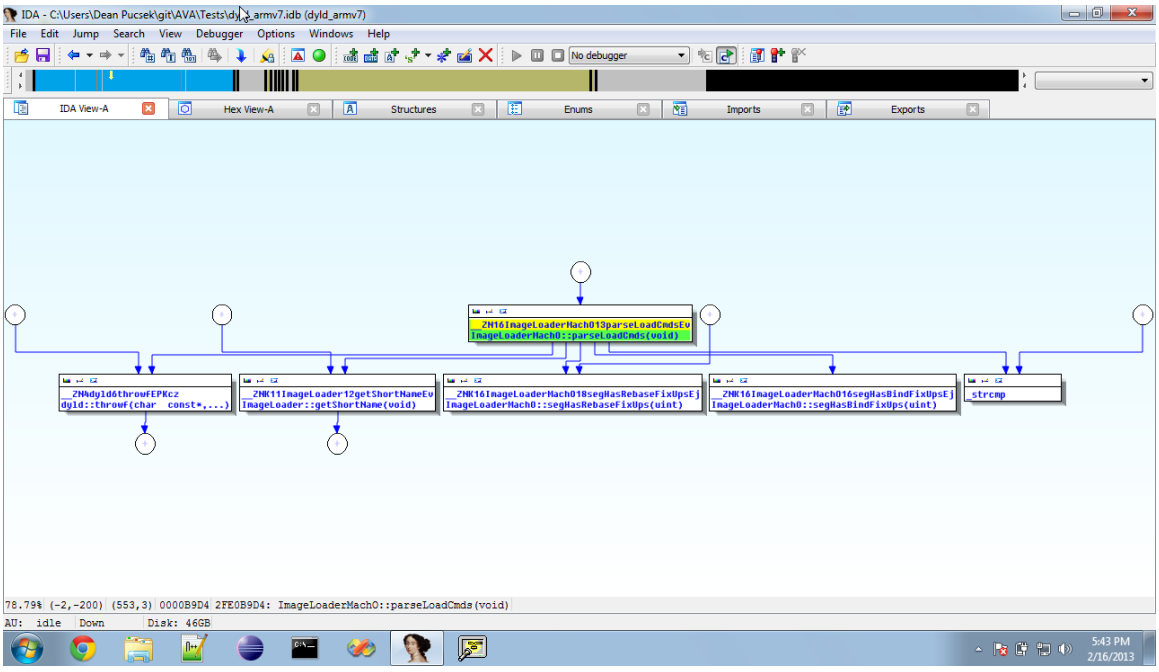

Figure 3.14: Proximity View of parseLoadCmds() in IDA Pro

analysis.

As a reference for this case study, the analysis provided [51] by the authors of the

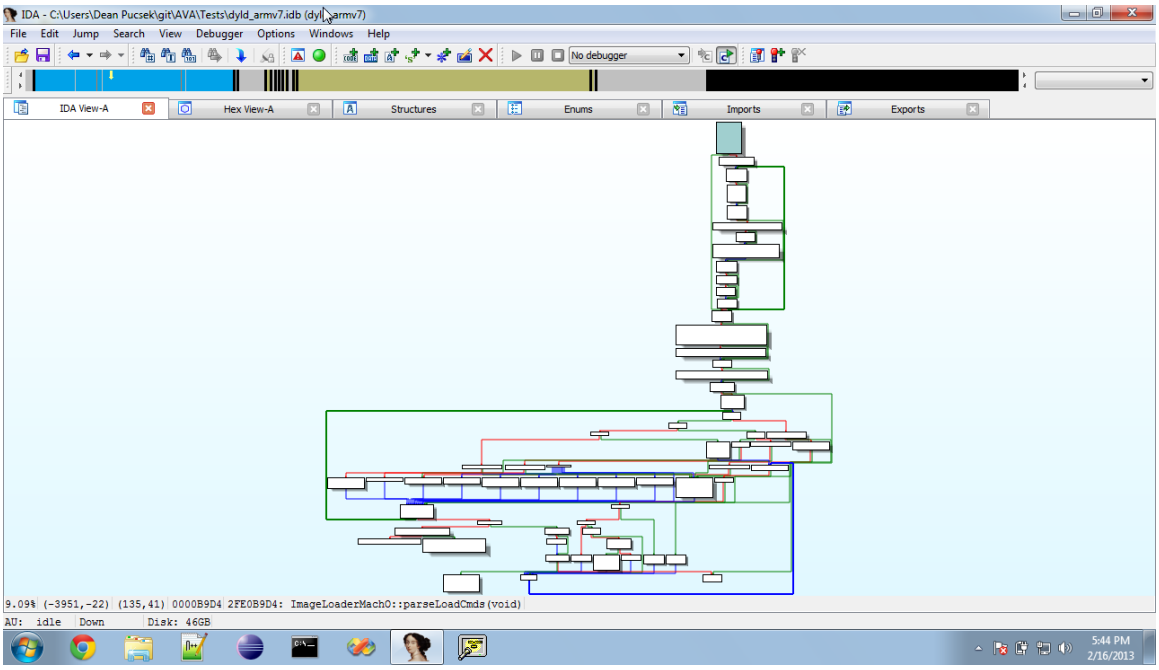

Figure 3.15: Control flow graph of parseLoadCmds() in IDA Pro

```
add         r1, sp, #0x3c
str         r1, [r4]
str         r5, [r4, #0x2c]
str         r0, [r4, #0x34]
add         r1, sp, #0x28
mov         r0, r5
str         r5, [sp, #0x1c]
bl          __ZN4dyld4loadEPKcRKNS_11LoadContextE ; dyld::load(char const*, dyld::LoadContext const&)
mov         r6, r0
add         r0, sp, #0x38
cmp         r6, #0x0
bne         0x2fe07960
; Basic Block Input Regs: r0 -  Killed Regs: r5 r6
movs        r6, #0x0
mov         r5, r0
b           0x2fe07a58
```

Figure 3.16: Disassembly of dlopen() in Hopper

sample is used to gauge to correctness and completeness of this study.

3.2.2 Initial Analysis

The first step taken during malware analysis is to conduct a static analysis on the binary as a whole and then to conduct a behavioural analysis as the sample is executed. The goal of this initial analysis is to gain insight into the intentions and functionality of the sample before the process of reverse engineering begins.

The initial static analysis of this sample was conducted using a variety of tools (Appendix B) and we found that the sample makes several references to a file named `msgina32.dll` and that a binary is contained within the resources section of the

Method	Functions Identified
Source Code	40
ICE	34
IDA Pro	30

Table 3.1: Summary of the functions identified in dlopen()

sample. Upon extracting the binary contained within the sample, we found that it is a dynamic link library (DLL) which is likely used as the replacement GINA [40] DLL.

A behavioural analysis was then conducted where the sample was observed during execution. During this analysis we observed that the sample makes no attempt to communicate over the network. We also saw that the sample extracts the DLL contained within it and modifies the Windows registry so that it is used as the GINA DLL as suspected. Ultimately, this sample allows Windows credentials to be harvested for later use.

3.2.3 Analysis with ICE

We begin the analysis by viewing the function calls made by the `main()` function in Cartographer 3.17.

From Cartographer it is seen that the malware sample makes two calls to code in the sample itself and the remainder to functions in Windows API. The two function calls to itself are `sub_401000` and `sub_401080`. Our analysis continues in `sub_401080`.

The call graph presented shows that the function makes calls to various functions in the Windows API related to loading and using resources as well as a single call to another function. Even without knowing the details of the function it is a fairly safe assumption that this function unpacks the DLL stored in the sample and writes it to file. At this point it is a good idea to rename the function so that it better describes what it does and assign a comment.

Moving deeper into the call stack, we encounter a function called `sub_401679` (Figure 3.18). From the call graph produced by Cartographer it is not clear what the intent of this function is. However, we are able to see that there is a strong correlation between this function and `get_int_arg` as well as `write_string` and `write_multi_char`. Since the intent of this function is not apparent from the information provided by Cartographer it is now necessary to analyze the assembly code directly in IDA Pro as in the first analysis.

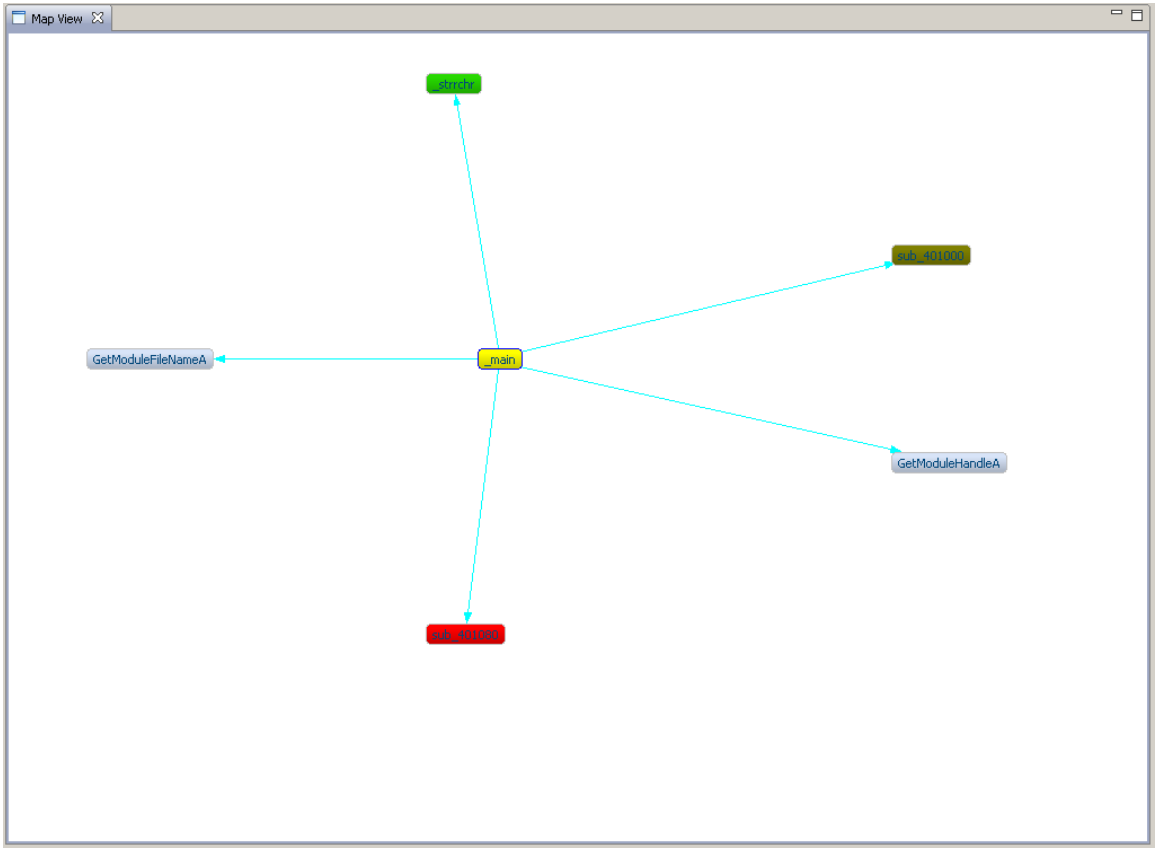

Figure 3.17: Call graph of main() produced by Cartographer

Since the binary contained a DLL in the resource section it is also necessary to examine it. The main entry point of a DLL is a function called `DllMain()`, this will be our starting point and is shown in Figure 3.19.

From Figure 3.19 it is difficult to grasp fully what is going on in this function. While is clear that the function gets the path of the system directory, loads a library, and does a string concatenation it is not clear what library it is trying to load and why. To learn this we would need to turn to the assembly code and work through the data being accessed.

Another point of interest about this malware is that it appears to implement all required functions for a GINA DLL replacement [4]. Moreover, as seen in Figure 3.20, each implementation of these functions calls another subroutine.

The function being called, `sub_10001000`, is shown in Figure 3.21. Here we see a call to `sprintf()`, `GetProcAddress()`, and `ExitProcess()`.

[4]The full list of functions required in GINA DLL can be seen at http://msdn.microsoft.com/en-us/library/windows/desktop/aa374731.aspx

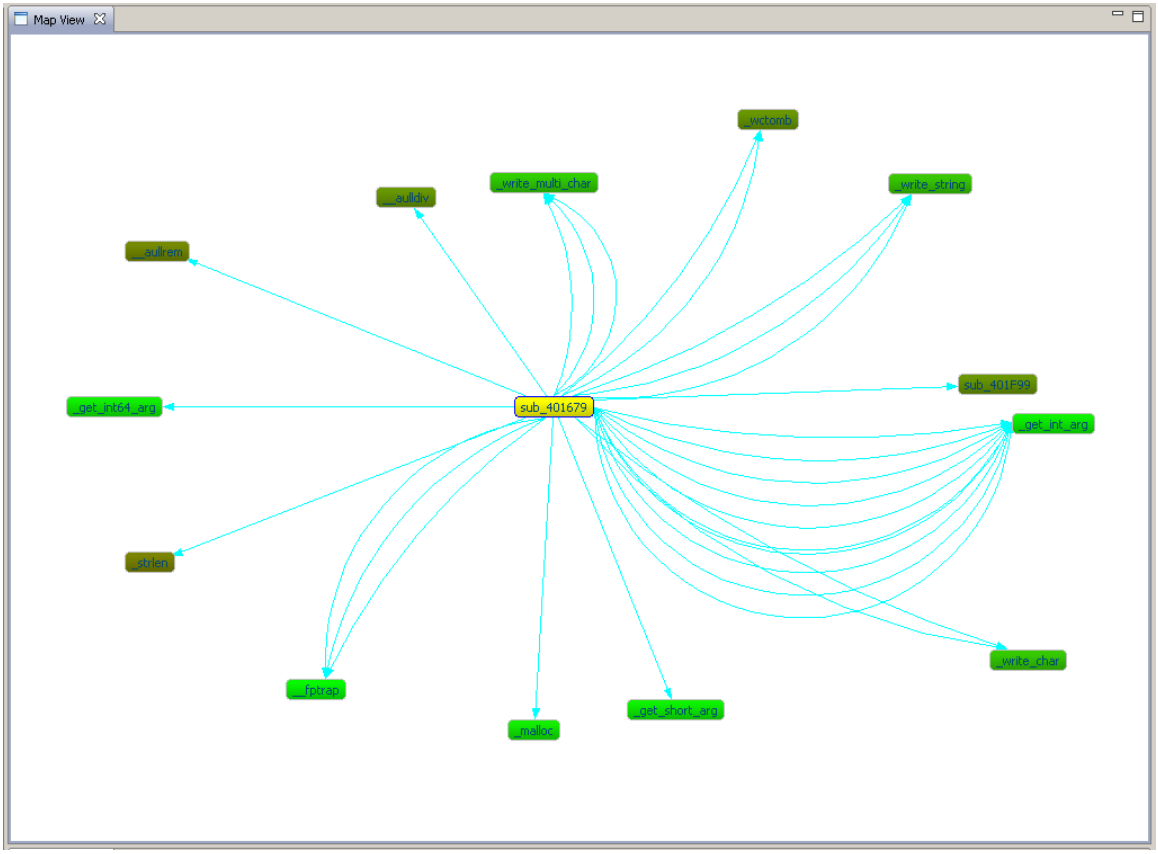

Figure 3.18: Call graph of sub_401679() produced by Cartographer

Since the intention of `GetProcAddress()` is to find the address of an exported function in a library it is likely that the library being loaded in `DllMain()` was official GINA library and that this usage of `GetProcAddress()` is where the malware jumps back to the official implementation so that users don't suspect any malicious activity. Note also the usage of `sprintf()` to print a message to a file, this is likely used to log passwords entered at the logon screen.

At this point we have covered both the main executable and the library dropped by this malware. Without having looked at a single assembly instruction we have information on the relationships between functions and even been able to take an educated guess at how the malware operates.

3.2.4 Analysis with IDA Pro

Analyzing this malware with IDA Pro proceeds as follows. Once again we begin by analyzing the main executable and then progress to the contained library.

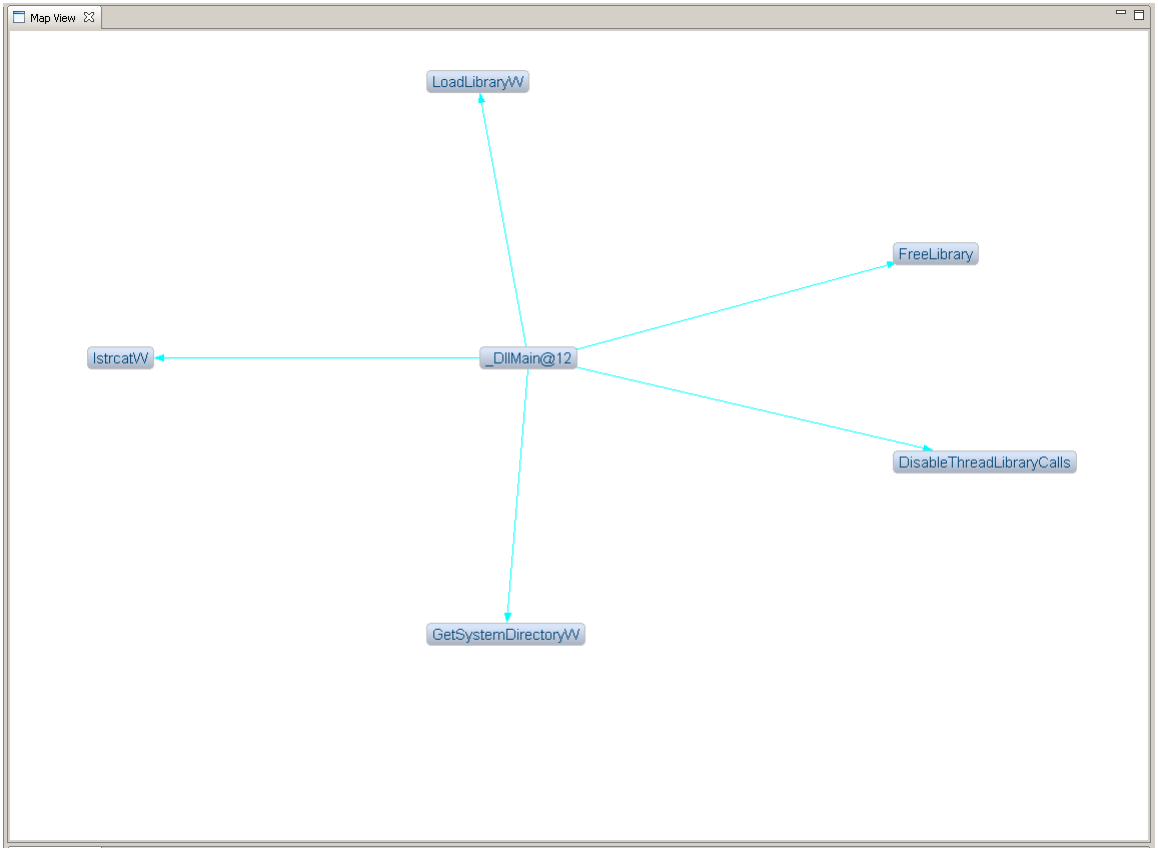

Figure 3.19: Call graph of DllMain() produced by Cartographer

Figure 3.22 shows the call graph of the main executable provided by IDA Pro. From this graph [5] it is seen that the `main` function makes a call to a function that deals with resources and one that makes calls to the API for the Windows registry. Although this is useful information, the graph is static and we are forced to search through IDA Pro for the `main` function.

At this point we focus our attention on the `main` function. Figure 3.23 is the *Proximity View* of this function. The Proximity View is a visualization provided by IDA Pro that shows all references, both code and data, to and from a function. For this analysis we use the Proximity View to see what functions `main` calls and what data it manipulates. With the Proximity View we once again see that `main` calls two functions in the sample in addition to some functions in the Windows API.

Turning our attention to the function `sub_401080()` we find, as we did with ICE, that this function is responsible for locating and extracting the DLL from the main

[5] Analysis was completed using a properly magnified graph

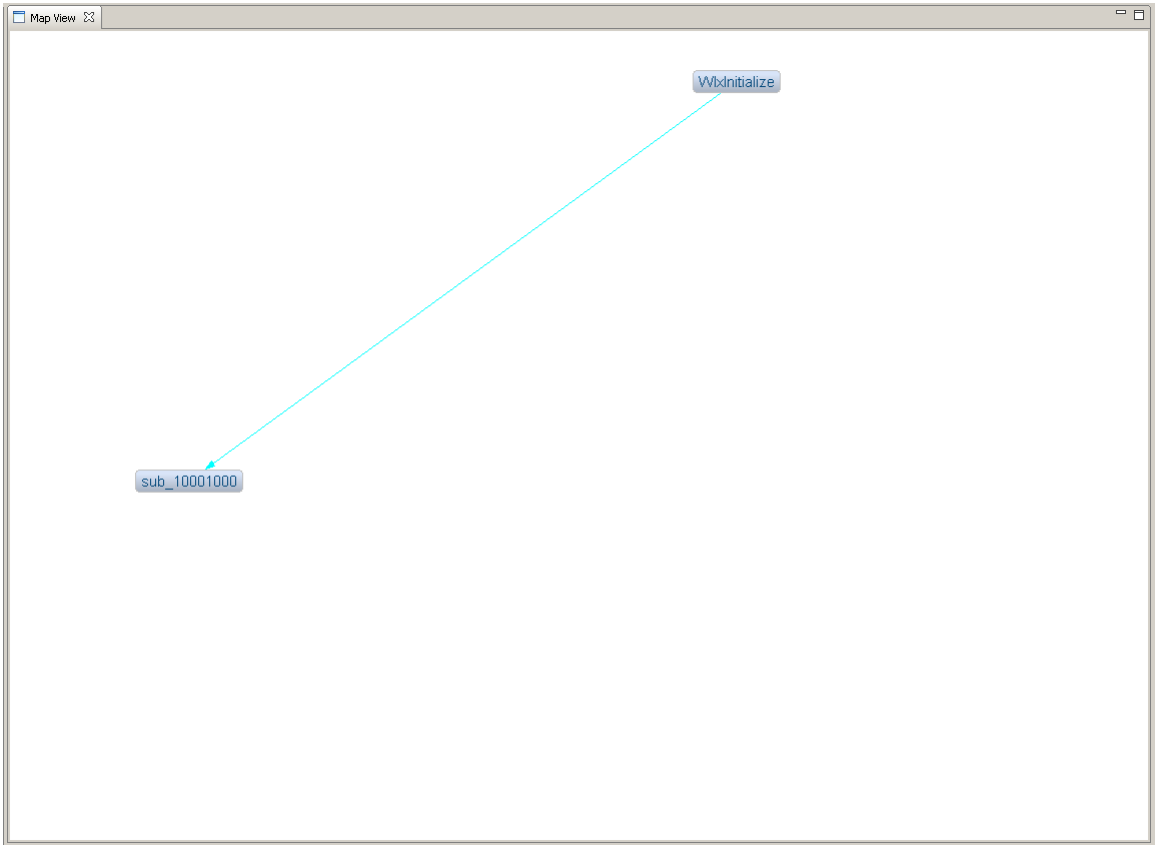

Figure 3.20: Call graph of WlxInitialize() produced by Cartographer

executable. If we now look at the function `sub_401000()` in IDA Pro we will see that it is responsible for retrieving and setting the value of a registry key. To determine which key is set and where the library is stored it is necessary to analyze the data being accessed in the assembly code.

A similar analysis can be carried out for the library that was extracted from the main executable. The information available and ease of the analysis is on par with that of ICE, in particular because the high level does not require the usage of a control flow graph.

3.2.5 Analysis with Hopper

Finally, completing the analysis with Hopper is a bit more of a challenge. As with both ICE and IDA Pro we want to start our analysis of the main executable in the `main()` function. Unfortunately, Hopper is not able to identify and label the function as main so it necessary to search manually.

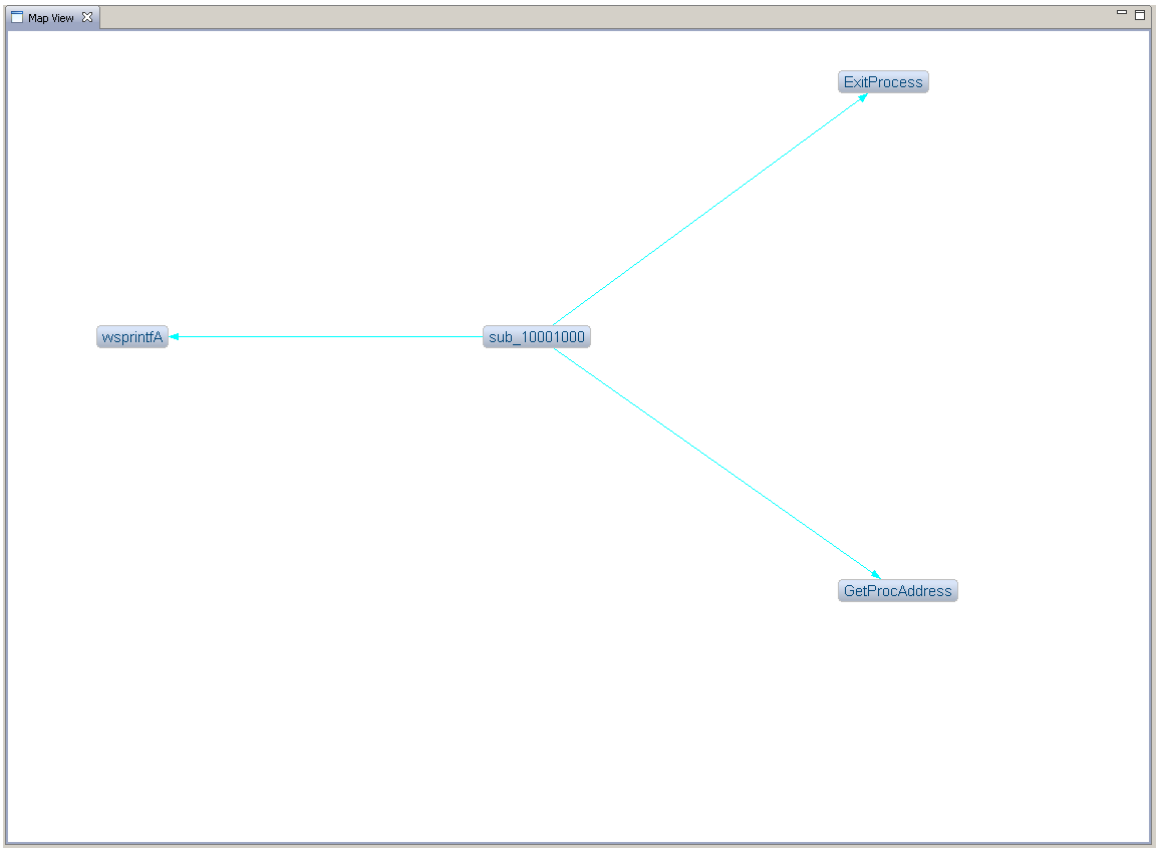

Figure 3.21: Call graph of sub_10001000() produced by Cartographer

The search can be conducted in one of two ways. First, Hopper does identify the initial entry point of the executable (commonly referred to as `start()`). So we could work through the assembly code and eventually identify the function that looks the most promising. Second, since we have previously carried out the analysis we can search for references to the functions `sub_401000()` and `sub_401080()`. Regardless of which method is used it is either necessary to (1) work directly with the assembly code or (2) leverage knowledge from a previous analysis. Since there is no visualizations in Hopper it is necessary to analyze the assembly code to determine the functionality of the code.

The process of working through assembly code continues with both `sub_401000()` and `sub_401080()`. However, due to the need to work with the assembly code it is a much slower process and may lead to an incorrect understanding of the code.

A similar approach can be taken to the library that was extracted from the main executable. The lack of visualizations in Hopper continues to impede comprehension and force the usage of the assembly code as the primary means to gaining insight into

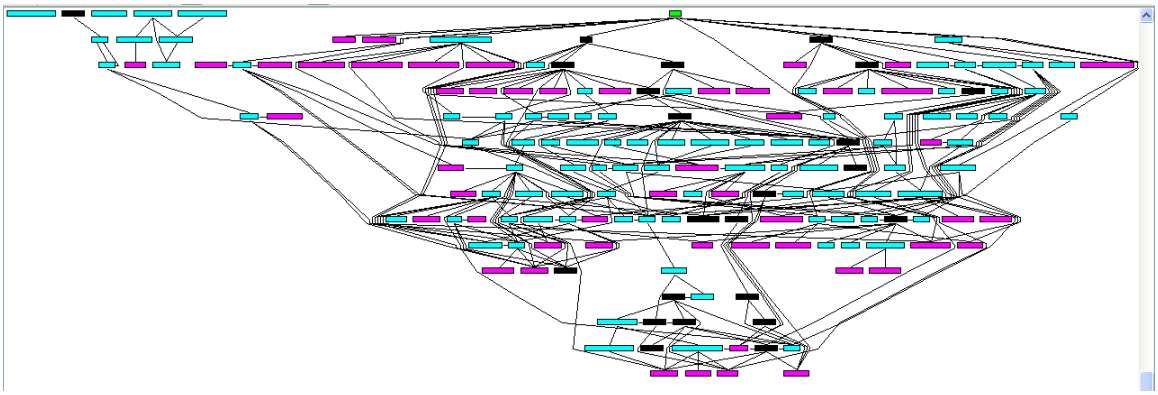

Figure 3.22: Function call graph of malware produced by IDA Pro

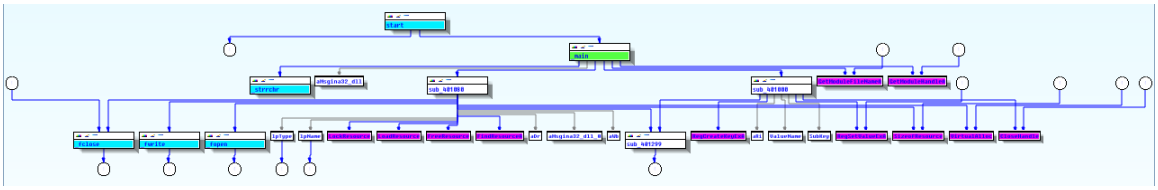

Figure 3.23: Proximity view of main() produced by IDA Pro

the functionality of the library.

3.2.6 Evaluation: Practical Malware Analysis

A detailed analysis of the malware sample was provided by the authors in *Practical Malware Analysis* [51] and can be found in Chapter 11 Lab 1 of the textbook.

The intention of the malware in this case study is to gather user credentials during the logon process through a mechanism known as GINA. When the malware is executed it extracts a library from its resource section named "msgina32.dll" and stores it on the local computer. The malware than modifies the GINA registry key pointing it to the newly extracted library.

The library simply implements the required functions by redirecting them to the Microsoft implementation in the original GINA library. For the functions that deal with user credentials they are written to a log file on the local system.

Both ICE and IDA Pro were able to successfully guide us to the correct conclusions about the malware where as Hopper required a significant amount of work to even identify the `main()` function before working through the assembly code to understand its functionality.

3.2.7 Sample Restrictions

The restrictions placed on the sample—no packing and no self-modifying code—were a result of trying to analyze the Mariposa [58, 52] botnet. During this analysis we found that the static nature of ICE made it extremely difficult to work with self-modifying code which is used for packing. Note that this is inherent in all static analysis techniques regardless of the tool used.

3.3 Case Study: Data Source Integration

The case studies outlined in Section 3.1.5 and Section 3.3.2 focused on different analyses that may be conducted in order to better understand how a piece of software works. This final case study shifts that focus and looks at how well ICE is able to work with multiple Data Sources and what it takes to integrate ICE into existing tools.

3.3.1 Multiple Data Sources

A common technique in software development is to leverage existing code as much as possible in order to cut down on development time and costs. Consequently, the use of libraries has become quite routine. Due to the ubiquity of libraries the need to be able to analyze the main executable as well as associated libraries is an aspect of program comprehension that must be considered.

As a demonstration of ICE and the default visualizations two open-source projects have been selected under the LLVM umbrella project for analysis. The main executable that will be analyzed is the *LLVM Debugger (LLDB)* and the associated library is the LLVM implementation of the standard C++ library, *libstdc++*. The goal of this analysis is to use ICE to trace through a specific feature of the debugger, the command-line option parsing, and then to drill down to investigate the implementation of a function in the standard C++ library.

The first step of our analysis is to look at the implementation of the `main()` function since this is the entry point into LLDB and where command-line options are first available. Figure 3.24 shows `main()` as seen in Cartographer; however, to gain an idea of the order in which function calls are made it is beneficial to look at `main()` as seen through Tracks (Figure 3.25). From these two views we know that the `main()` function begins by initializing the `lldb::SBDebugger` object, spawns a

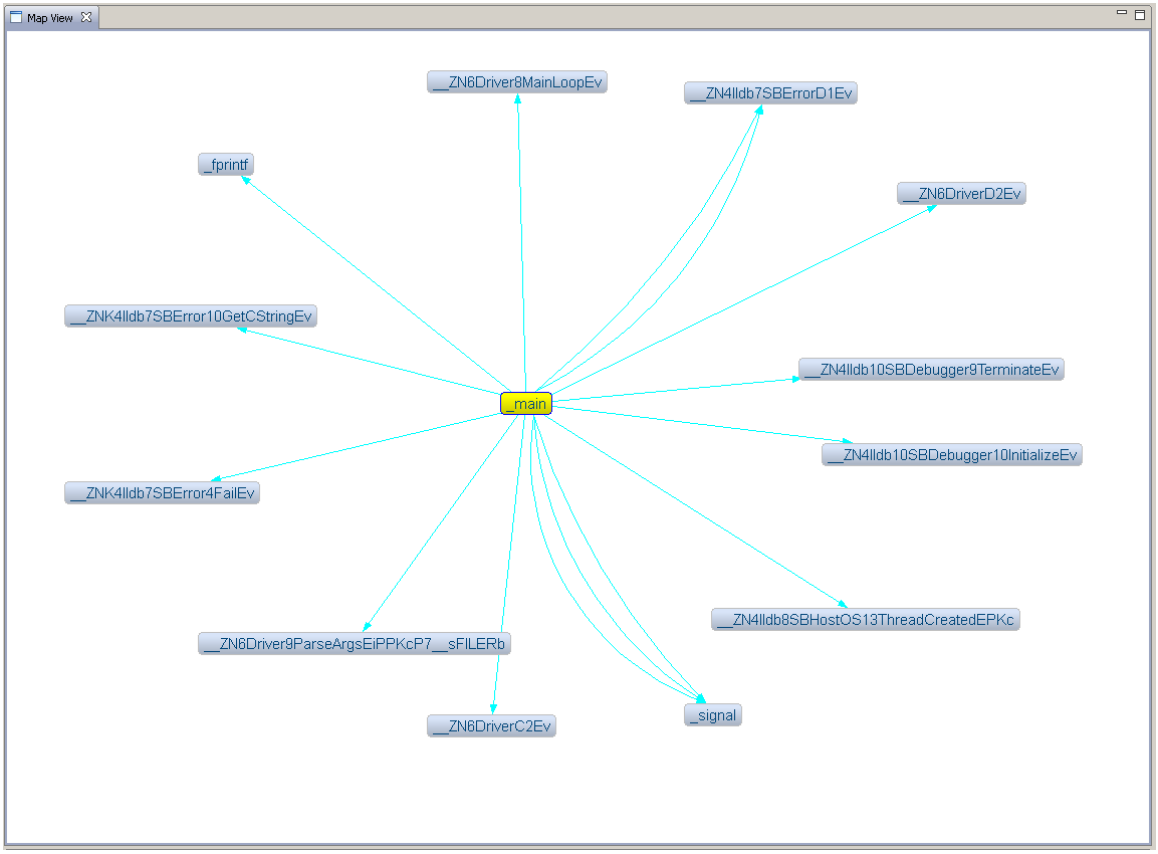

Figure 3.24: Call graph of LLDB's main() function produced by Cartographer

thread, modifies signal handlers, parses command-line arguments, and then enters the main loop of LLDB.

This combination of Cartographer and Tracks reveals that command-line arguments are passed to a function called `Driver::ParseArgs()`. Double-clicking this entry will enable Cartographer to display the call graph for `Driver::ParseArgs()`, Figure 3.26, which reveals that a large number of functions are called from here. Due to the lack of data-flow in this view it is not possible to determine which command-line arguments result in which function calls; but, it is clear that the arguments result in functions being called that can set the architecture to be debugged and query the version of LLDB among other things. One point of interest in the Cartographer visualization is that the number of lines connecting two functions indicates the number of times the function is called.

One function call in Figure 3.26 that stands out is a call to function called `push_back()`. Although the function displayed contains the C++ name mangling

from the compiler, this can be decoded to learn that this function is part of the standard C++ library string implementation. This can be confirmed by searching for `push_back` in the *Function Selector* of Cartographer as seen in Figure 3.27.

At this point we can leverage the Control Flow Graph visualization in ICE to get a better idea of how the `push_back()` function is implemented. Figure 3.28 shows the CFG of `push_back()` with Joins highlighted. Joins are any point in a function where an instruction has either multiple predecessors or successors and in this case shows that `push_back()` likely contains nested if-else statements.

The ability to work with multiple data sources in ICE enables not only the collection of data from a variety of sources such as IDA Pro, debuggers, and even IDEs; but it also enables the data to be consolidated into a single model that can be analyzed through the visualizations provided.

3.3.2 Data Source Integration

Currently the primary tool used in conjunction with ICE for disassembly is IDA Pro. However, as an experiment to get a better understanding of how well the ICE communication model suits other tools and to demonstrate that ICE is not explicitly tied to IDA Pro a program based on a popular disassembling library was written that could interact with ICE.

Writing a custom program was selected as the approach for this experiment because of lack of a plugin system provided by other disassemblers. For example, Hopper—used in the previous case studies—provides a minimal scripting environment; however, since this is a Python interface and the Python interpreter is embedded into the same process as Hopper executing long-lived scripts will cause Hopper to block. The situation is similar for popular debuggers such as OllyDBG and Immunity Debugger.

The program, *iced* [6], was written in Python and leverages the diStorm [7] disassembler library for the x86 instruction set. In addition to diStorm, iced also leveraged the standard Python socket library for communication as well as a library called pefile [8] to parse the PE file format for input to diStorm. Development of iced consumed approximately one day of work and resulted in fully functional connection with ICE.

With respect to the communication mechanism in ICE we found that the use

[6] Source code presented in Appendix A

[7] diStorm is an open-source project hosted at https://code.google.com/p/distorm/

[8] pefile is an open-source project hosted at https://code.google.com/p/pefile/

of sockets enabled a programming language agnostic approach. This is clearly seen given that the code for iced was written in Python, the code for IDA Pro in C++, and the code for ICE in Java. Extrapolating from this, given the near ubiquitous support for sockets in modern programming language environments, it is reasonable to conclude that ICE will be able to communicate with Data Sources of all varieties. A similar observation was made regarding the selection of JSON as the transport protocol between ICE and its Data Sources.

Through iced we found that the data required by ICE can be readily available given a suitable processing of the binary to be analyzed prior to communicating with ICE. This was first seen when passing information about the functions present in the binary to ICE from iced. Unlike IDA Pro, diStorm only provides a stream of disassembled instructions that must then be processed to identify function boundaries. During this experiment only 32-bit x86 binaries were used and we assumed that the binary correctly followed the System V ABI [56] which allowed for the identification of function boundaries. Providing information to ICE regarding call sites was less involved since the call instruction contains the address of target location. Finally, when producing a Control Flow Graph in ICE it was a straight forward process to iterate through all relevant instructions and send the data to ICE.

3.4 Summary

This chapter developed three case studies focusing on different aspects of program comprehension. The first case study investigates how one might use ICE to validate the implementation of an algorithm. The second case study looks at program comprehension of malicious software and, finally, the third case study looks at the ability of ICE to be integrated into existing Data Sources.

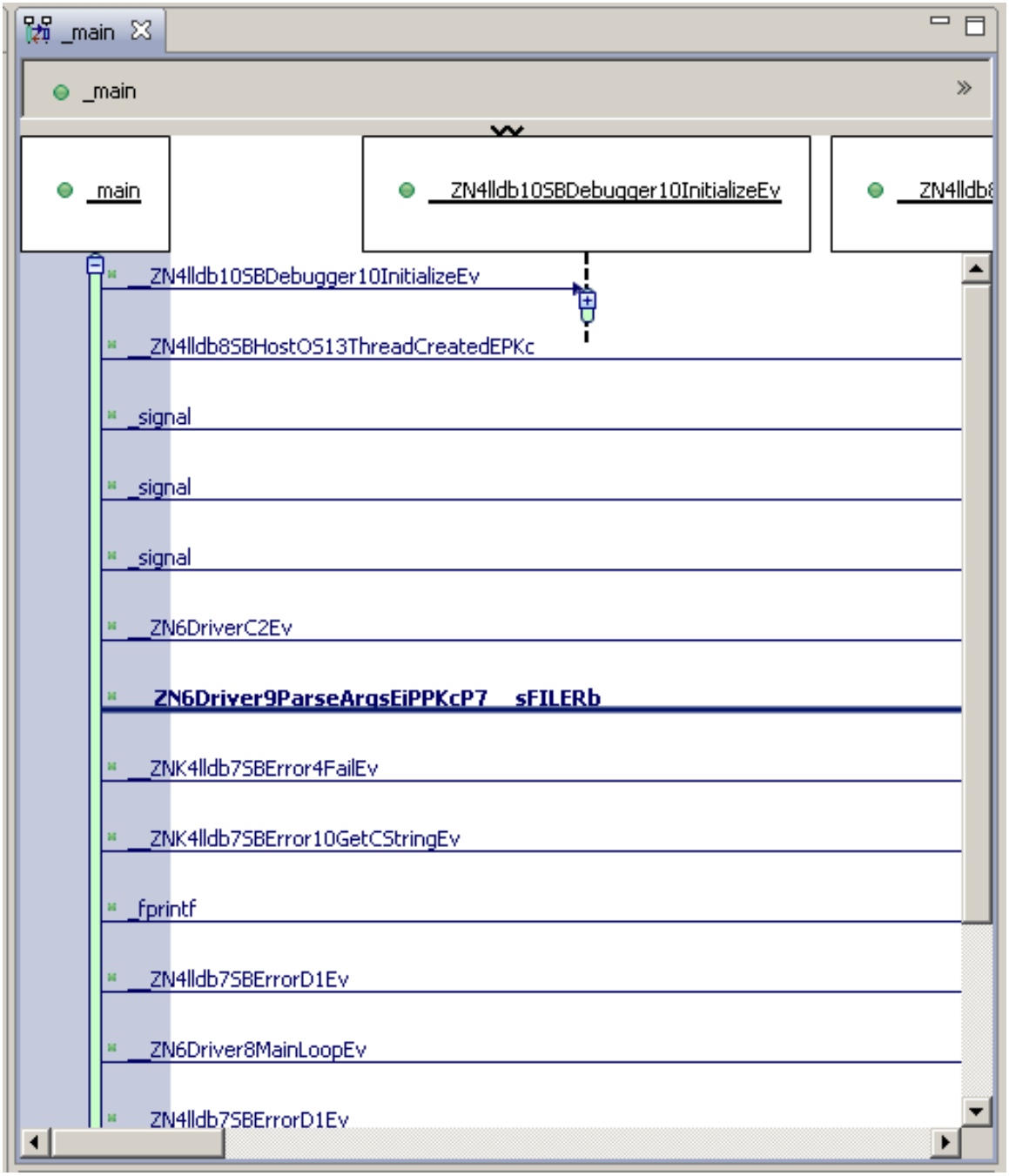

Figure 3.25: LLDB's main() as seen through Tracks

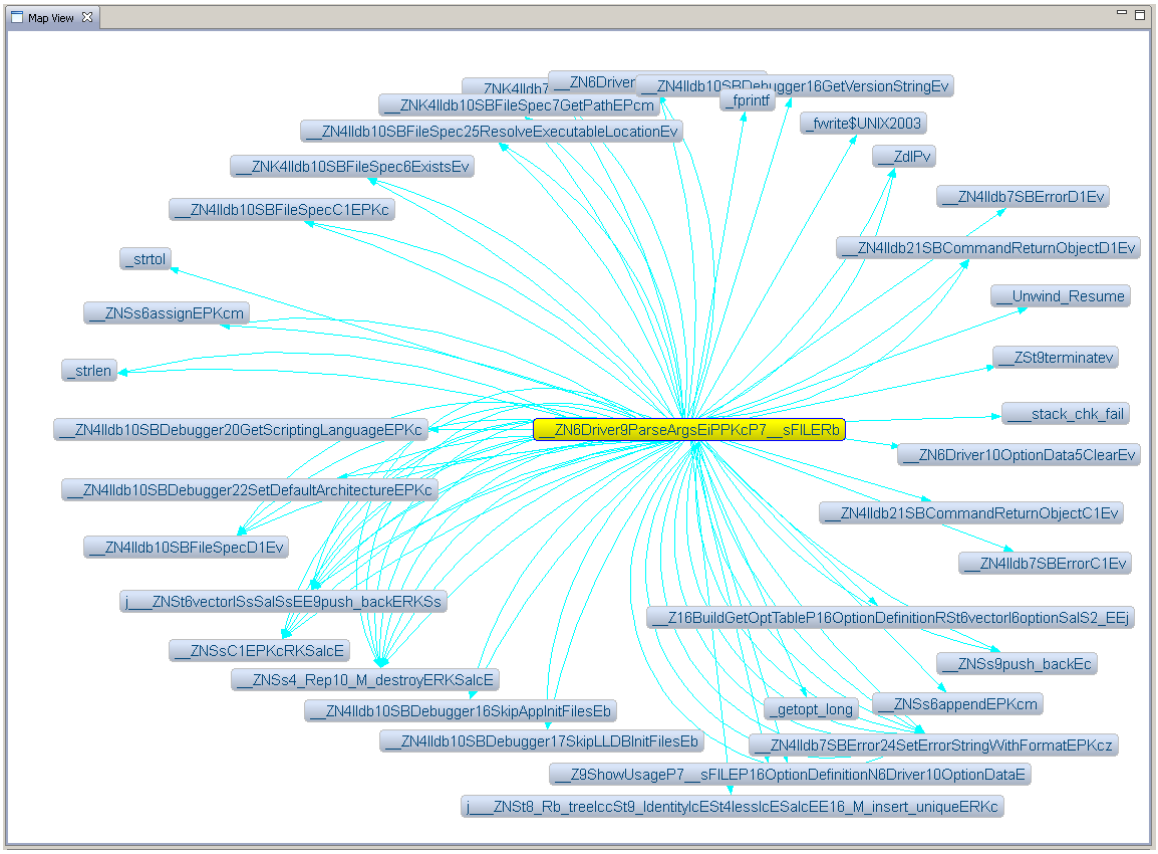

Figure 3.26: Screenshot of Driver::parseArgs() in Cartographer

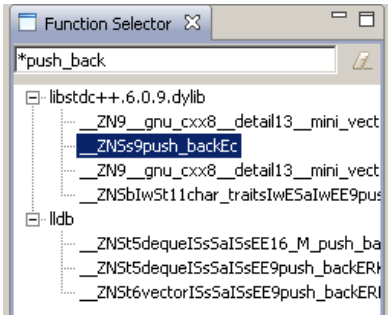

Figure 3.27: Searching for push_back() in Cartographer

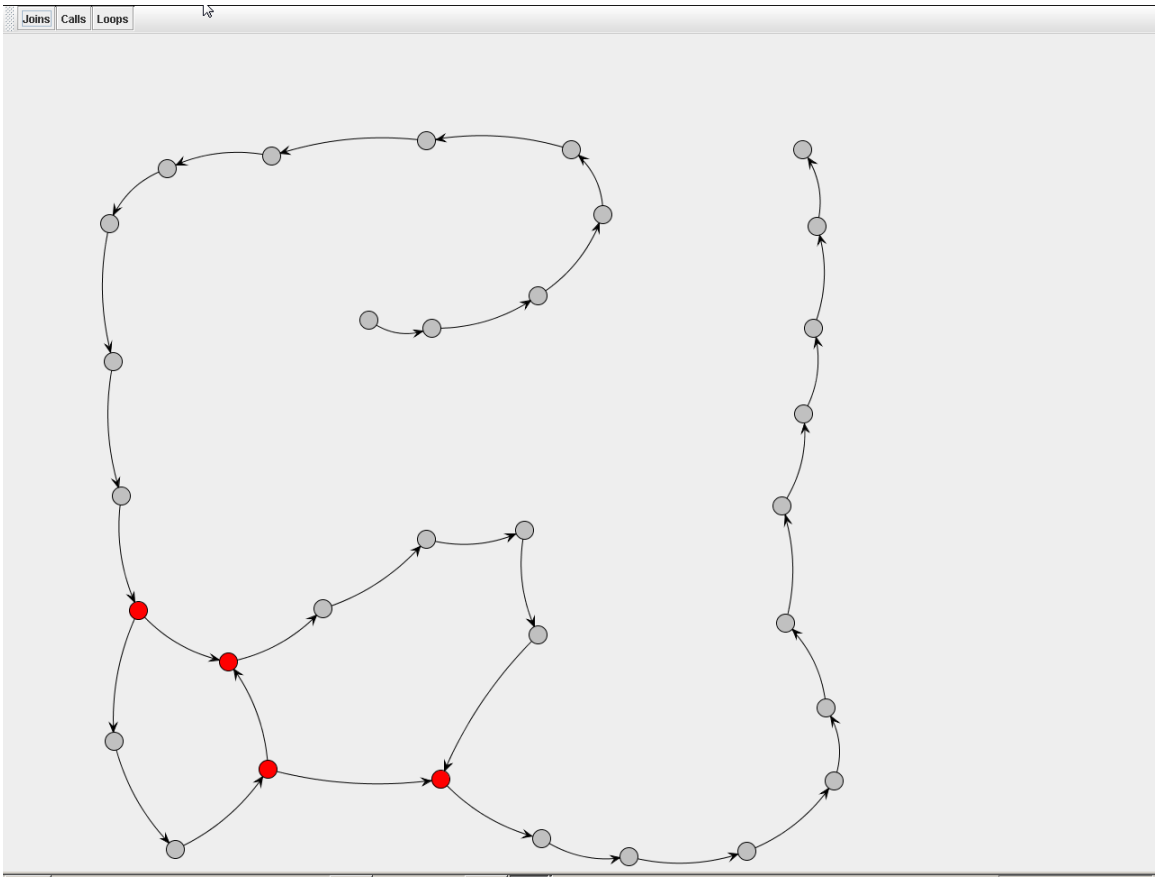

Figure 3.28: Control flow graph of push_back() with Joins highlighted

Chapter 4

Validation

This chapter evaluates ICE relative to the requirements identified in Chapter 1 as well as discusses specific aspects of ICE and the limitations it has with respect to program comprehension.

4.1 Validating the Requirements

The results of the experiments in Chapter 3 show that ICE is capable of meeting the initial needs outlined in Chapter 1 through its modular design and implementation as summarized in Table 4.1.

Requirement	Satisfying Component
Operate on multiple binaries	ICE Core
Cross references between binaries	ICE Core
Map of the analysis	Tracks, Cartographer and Tours
Inline documentation through tagging	TagSea

Table 4.1: Summary of requirements met by ICE

Of these requirements, the need to be able to analyze multiple binaries at one time as well as have data propagated between binaries are met due to ICE being grounded on the MVC model and demonstrated in Section 3.3.2. Furthermore, Figure 4.1 demonstrates the use of Tours on the same codebase as in Section 3.3.2.

Through the graph-based Data Model discussed in Section 2.4.2 ICE consolidates the various pieces of information that describe an Executable Entity as a single model accessible through visualizations. ICE provides the ability to track an analysis through two of the default visualizations, Tracks and Cartographer, and through

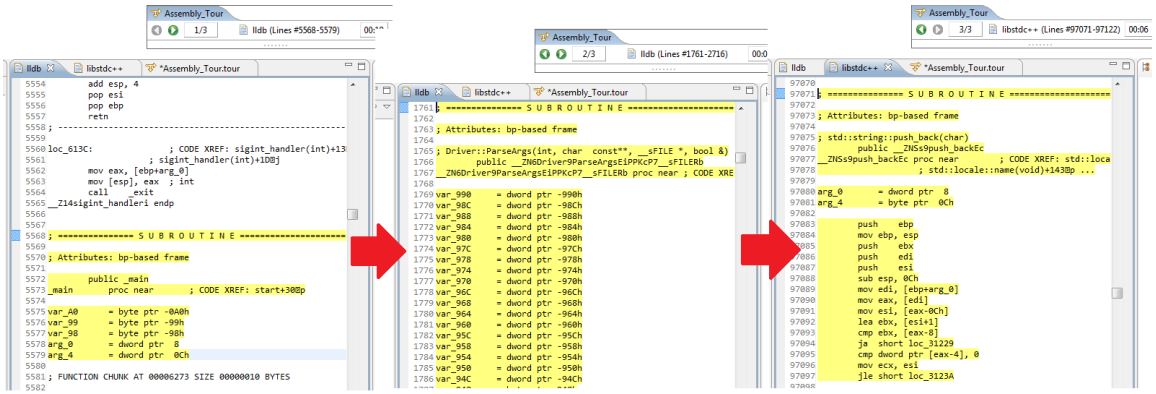

Figure 4.1: Tour of the LLDB and libstdc++ code

Tours. Finally, documentation can be created using TagSEA.

4.2 Ramifications of the Design and Implementation

In addition to meeting the requirements outlined in Chapter 1 there are several ramifications of the design and implementation of ICE that should be considered.

The primary benefit of the design and implementation of ICE is the flexibility and extensibility realized through the symbiosis of Eclipse, the graph-based Data Model, and the loosely coupled communication mechanism. The flexibility afforded by these technologies permeates ICE, enabling analysts to work with multiple Executable Entities simultaneously, and integrate visualizations that focus on the many levels of relationships between the levels of abstraction.

The Eclipse-based implementation of ICE is also able to support rapid development of new visualizations, just as Eclipse supports plugins. For example, the initial development of Cartographer took approximately two days of development time; including the time necessary to become familiar with the Zest rendering library. Similarly, refactoring Tracks and Tours took under a day to configure for the ICE environment. Due to the age of the Tours project, the main challenge with integrating this tool came from finding the correct version of multiple build files.

Currently the design and implementation of ICE assume that the Data Source will perform an analysis of its Executable Entity to identify function boundaries. The consequence of this assumption is that any piece of code not contained in a function as

identified by the Data Source will not be included in ICE. When developing ICE, we decided that this was a reasonable assumption for a prototype implementation as our primary Data Source in testing was IDA Pro which is capable of identifying functions and passing that data to ICE. In addition, the majority of binaries used for testing were not malicious in nature and were compiled using standard compilers. However, this assumption potentially becomes a limitation when working with malware and binaries that leverage some programming language features. With respect to programming language features, Data Sources are typically unable to identify to location of functions that been inlined as the generated assembly code is placed inside all calling functions. This is similar to the situation found in some malware where functions are malformed and do not follow a published calling convention. Another problem area is the analysis of exception handlers. Exception handlers are typically written such that they have multiple entry and exit points; however, the Data Model in ICE does not track entry or exit points.

Continuing with the discussion of multiple exit points, it is important to revisit the information ICE stores in the Data Model. From the discussion of the Data Model in Section 2.4.2 it is seen that for each function ICE stores an 'end location'. However, there is an important distinction between 'exit point' and 'end location' and it is that the 'end location' is the the last address that belongs in a given function where as an 'exit point' is any instruction that would cause execution to leave the function. The consequence of this distinction is that ICE does not track exit points in the Data Model. Conversely, when viewing a CFG through ICE it is possible to see multiple exit points because the CFG is generated using data provided directly from the instructions and not data from any previous analysis performed by a Data Source.

4.3 Limitations of ICE

Despite the many advantages of ICE it does bring some limitations to light. Firstly, ICE is currently incapable of performing dynamic analysis. This limitation will likely be a large issue for malware analysis as malware typically includes self-modifying code which must be evaluated in order to reverse engineer. During the development of ICE we consciously decided to leave dynamic analysis as future work.

Secondly, a big limitation of ICE in its current implementation is the reliance on an external application to disassemble and analyze the binary for function boundaries.

This becomes a problem because ICE must trust that the external application is providing the correct information and ICE is not able to supplement the information with an additional analysis. Furthermore, this approach of using external tools for disassembly and analysis ties ICE to the platforms supported by the tools being used. For the case studies conducted IDA Pro was used as the external tool which has an excellent reputation for disassembling and analyzing code; however, it would be beneficial to ICE to be able to perform these tasks as that would enable a larger range of Data Sources to be used.

Thirdly, the lack of data flow analysis present in ICE limits an analyst to working either at the function-level or with the control flow of an Executable Entity. As an example, this limitation would prevent an analyst from performing an analysis that identifies how user-input modifies the internal state of the program.

In addition to the limitations that can be attributed to the core components of ICE there are a couple limitations in the default visualizations.

Tracks. The primary limitation of Tracks is that function calls are listed in ascending order based on the address of the call. The issue with this approach is that the analyst can be mislead to believe one function call occurs before another when, potentially, it may occur after due to control flow used in the function.

Control Flow Graph. The Control Flow Graph (CFG) facility provided through ICE provides an analyst with the opportunity to gain insight into the implementation of a function without reading the assembly code. However, due to the typically high number of instructions required to implement a function the CFG can become extremely cluttered leading to information overload. The filters provided help mitigate the information overload; however, as indicated in the future work, more needs to be done to help an analyst grasp the implementation of a function.

4.4 Summary

This chapter presented an evaluation of ICE relative to requirements of Chapter 1 where we found that ICE meets the requirements through its core components and

visualizations. Additionally, a discussion of the consequences of the design and implementation was presented along with the known limitations of ICE.

Chapter 5

Future Work and Conclusions

This chapter concludes the thesis and provides suggestions for future work to improve ICE and better understand its impact on program comprehension.

5.1 Conclusion

Framework	Foundation	Extensibility Mechanism	Visualizations
ICE	Hybrid	Communication and core architecture	Yes, extensible
IDA Pro	Binary	Plugins (SDK)	Yes, not extensible
BinNavi	Binary	None	Yes, not extensible
BitBlaze	Binary	None	No, not extensible
LLVM Code	Intermediate Language	API, Libraries	No, not extensible

Table 5.1: Summary of frameworks relative to ICE

Program comprehension is an extremely difficult task where relationships between components, and the manner in which information flows through the program, must be reasoned about at both a high- and low-level in many codebases. The proposed Integrated Comprehension Environment provides a framework for analysts to visualize and share the relationships they find in a program through the use of interactive call graphs and sequence diagrams. ICE is specifically designed to be flexible and extensible, allowing new visualizations to be developed and shared as they are required. We have explored how ICE meets many of the needs in this domain, how it can easily accept new sources, and how it can be used to mitigate issues of scale at this level

where the number of instructions is formidable compared to high-level code. Table 5.1 provides a summary of how ICE compares to existing program comprehension solutions. Though ICE establishes a good first step as a prototype comprehension environment in this domain, it still has several limitations (Section 4.3), which we plan to address in future work.

5.2 Future Work

The most pressing item in terms of future work is to extend ICE such that it is able to support dynamic analysis. ICEs inability to support dynamic analysis is not a limitation of the design and was intentionally left out due to the engineering effort required. In order to provide support for dynamic analyses Data Sources would need to notify ICE when new data is available. Not only would this enable dynamic analysis of code it would also open up the possibility of investigating hybrid static-dynamic analysis techniques.

Furthermore, development time is planned to provide support for an interface to the Data Model through a query language. This interface would enable analysts to focus on specific characteristics of the data available—performing queries such as *what function calls this API?* and *find all functions that implement this characteristic.* A query language would also find great use in overloaded and cluttered visualizations.

Another consideration, when working with object-oriented codebases, is that it is currently challenging to associate functions with the containing class. Additionally, the lack of a class diagram leaves some questions surrounding the code base.

With respect to avenues for research, it would be interesting to perform a user study and investigate a potential normalized representation of instructions. Through a user study it would be possible to better understand how ICE can fit into an analysts workflow, what cognitive barriers—either broken or found—may be associated with ICE, and to better understand what limitations may be associated when working with code written using a variety of programming paradigms. Finally, investigating possible representations of instructions would allow ICE to be further separated from the underlying instruction set as well as enable it to include functionality to discover functions or other aspects of a program.

Bibliography

[1] objdump: Gnu binary utilities. http://www.gnu.org/software/binutils/. Accessed on: 04-12-2012.

[2] otool - object file displaying tool. https://developer.apple.com/xcode/. Accessed on: 04-12-2012.

[3] F. E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[4] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. Wysinwyx: What you see is not what you execute. *Verified Software: Theories, Tools, Experiments*, pages 202–213, 2008.

[5] J. Baldwin, P. Sinha, M. Salois, and Y. Coady. Progressive user interfaces for regressive analysis: Making tracks with large, low-level systems. In *AUIC'11: Proceedings of the Australasian User Interface Conference*, 2011.

[6] Jennifer Baldwin and Yvonne 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, pages 17–24, New York, NY, USA, 2011. ACM.

[7] Jennifer Baldwin, Alvin Teh, Elisa Baniassad, Dirk van Rooy, and Yvonne Coady. Applying social psychology techniques to requirements elicitation within highly-specialized industry software groups. *In Submission*, 2013.

[8] Jon Beck and David Eichmann. Program and interface slicing for reverse engineering. In *Proceedings of the 15th international conference on Software Engineering*, pages 509–518. IEEE Computer Society Press, 1993.

[9] Fabrice Bellard. Qemu: A fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–46, 2005.

[10] C. Bennett, D. Myers, M. A Storey, and D. 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)*, pages 1–5, Vancouver, Canada, 2007.

[11] D. Callahan, A. Carle, M. W. Hall, and K. Kennedy. Constructing the procedure call multigraph. *Software Engineering, IEEE Transactions on*, 16(4):483–487, 1990.

[12] Li-Te . T. Cheng, Michael Desmond, and M-A . A. Storey. Presentations by programmers for programmers. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 788–792. IEEE, 2007.

[13] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[14] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to high-level language translation. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 228–237. IEEE, 1998.

[15] Codeviz: A callgraph visualiser. http://www.csn.ul.ie/ mel/projects/codeviz/. Accessed on: 2012-12-07.

[16] cscope: Source code browser. http://cscope.sourceforge.net/. Accessed on: 2012-12-07.

[17] ctags. http://ctags.sourceforge.net/. Accessed on: 2012-12-07.

[18] Brumley D and Jager I. The bap handbook. 2009.

[19] R. DeLine. Staying oriented with software terrain maps. In *Proc. of the Workshop on Visual Languages and Computation*. Citeseer, 2005.

[20] Isil Dillig, Thomas Dillig, and Alex Aiken. Sail: Static analysis intermediate language with a two-level representation. Technical report, Stanford University Technical Report, 2009.

[21] diStorm - Powerful Disassembler Library for x86/AMD64. http://code.google.com/p/distorm/. Accessed on: 2012-12-07.

[22] S. Ducasse, T. Girba, and A. Kuhn. Distribution map. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*, pages 203–212. IEEE, 2006.

[23] T. Dullien and S. Porst. Reil: A platform-independent intermediate representation of disassembled code for static code analysis. *CanSecWest*, 2009.

[24] The eclipse foundation. http://eclipse.org/. Accessed on: 2012-12-07.

[25] Eclipse rich client platform. http://www.eclipse.org/home/categories/rcp.php. Accessed on: 2012-09-10.

[26] Peter Ferrie. Anti-unpacker tricks. In *Virus Bulletin*, page 4, 2008.

[27] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina. Smartdec: Approaching c++ decompilation. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 347–356. IEEE, 2011.

[28] GDB: GNU Debugger. http://www.gnu.org/software/gdb/.

[29] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *ACM SIGPLAN Notices*, volume 32, pages 108–124. ACM, 1997.

[30] Ilfak Guilfanov. Decompilers and beyond. *BlackHat USA 2008*, pages 1–12, 7 2008.

[31] Haoran Guo, Jianmin Pang, Yichi Zhang, Feng Yue, and Rongcai Zhao. Hero: A novel malware detection framework based on binary translation. In *Intelligent Computing and Intelligent Systems (ICIS), 2010 IEEE International Conference on*, volume 1, pages 411–415. IEEE, 2010.

[32] Hopper - disassembler for mac os x.

[33] IDA Pro. http://www.hex-rays.com/products/ida/index.shtml. Accessed on: 2012-09-10.

[34] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Modeling Language*. University Video Communications, 1996.

[35] JSON: JavaScript Object Notation. http://www.json.org/. Accessed on: 2012-12-07.

[36] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, pages 255–270, 2004.

[37] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, 3 2004.

[38] LLVM Language Reference. http://llvm.org/docs/LangRef.html Accessed on: 2011-08-20.

[39] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. Testing cpu emulators. In *Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA)*, pages 261–272, Chicago, Illinois, USA. ACM.

[40] Microsoft. Winlogon and gina. http://msdn.microsoft.com/en-us/library/windows/desktop/aa380543.aspx.

[41] Hausi A. Müller, Scott R. Tilley, Mehmet A. Orgun, B. D. Corrie, and Nazim H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *ACM SIGSOFT Software Engineering Notes*, volume 17, pages 88–98. ACM, 1992.

[42] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[43] OllyDbg. http://www.ollydbg.de/. Accessed on: 2011-08-20.

[44] Jianmin Pang, Yichi Zhang, Chao Dai, Di Sun, and Qiang Wang. A novel disassemble algorithm designed for malicious file. *Research Journal of Applied Sciences*, 5, 2013.

[45] Sukanya Ratanotayanon, Susan Elliott Sim, and Rosalva Gallardo-Valencia. Supporting program comprehension in agile with links to user stories. In *Agile Conference, 2009. AGILE'09.*, pages 26–32. IEEE, 2009.

[46] Trygve Reenskaug. Models-views-controllers. *Technical note, Xerox PARC*, 32:55, 1979.

[47] Trygve Reenskaug. Thing-model-view-editor-an example from a planningsystem. *Xerox PARC technical note*, 12, 1979.

[48] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. How do professional developers comprehend software? In *Proceedings of the 2012 International Conference on Software Engineering*, pages 255–265. IEEE Press, 2012.

[49] B. G. Ryder. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on*, (3):216–226, 1979.

[50] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H. Paterson. An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE working group reports*, pages 65–86. ACM, 2010.

[51] Michael Sikorski and Andrew Honig. *Practical Malware Analysis*. No Starch Press, 1 edition, February 2012.

[52] P. Sinha, A. Boukhtouta, V. H. Belarde, and M. Debbabi. Insights from the analysis of the mariposa botnet. In *Risks and Security of Internet and Systems (CRiSIS), 2010 Fifth International Conference on*, pages 1–9. IEEE, 2010.

[53] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. *Information Systems Security*, pages 1–25, 2008.

[54] M. A. Storey, C. Best, and J. Michand. Shrimp views: An interactive environment for exploring java programs. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 111–112. IEEE, 2001.

[55] Storey, M.A. and Cheng, L.T. and Singer, J. and Muller, H. and Myers, D. and Ryall, J. How Programmers Can Turn Comments Into Waypoints For Code

Navigation. In *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pages 265–274. IEEE, 2007.

[56] System v application binary interface: Intel 386 supplement. Technical report, 3 1997.

[57] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[58] M. Thompson. Mariposa botnet analysis. Technical report, Technical report, Defence Intelligence, 2009.

[59] Tiobe programming language index. http://www.tiobe.com/index.php/content/paperinfo/tpci Accessed on: 2011-08-20.

[60] P. Trinius, T. Holz, J. Gobel, and F. C. Freiling. Visual analysis of malware behavior using treemaps and thread graphs. In *Proceedings of the IEEE 6th International Workshop on Visualization for Cyber Security*, pages 33–38. IEEE, 2009.

[61] K. Troshina, A. Chernov, and Y. Derevenets. C decompilation: Is it possible? In *Proceedings of International Workshop on Program Understanding*, pages 18–27, 2009.

[62] Microsoft visual studio. http://www.microsoft.com/visualstudio/.

[63] Anneliese von Mayrhauser and A. Marie Vans. From program comprehension to tool requirements for an industrial environment. In *Proceedings of IEEE Second Workshop on Program Comprehension*, pages 78–86. IEEE, 1993.

[64] Mark V. Yason. The art of unpacking. In *BlackHat USA 2007*, volume 12, page 2008, 7 2007.

[65] Eclipse zest - graphing framework. http://www.eclipse.org/gef/zest/.

[66] Zynamics. BinNavi. http://www.zynamics.com/binnavi.html.

Appendix A

Source Code Listing for iced

```
# file: iced.py
#
# author: Dean Pucsek <dpucsek@uvic.ca>
# license: none, free in all senses of the word
#
# Script to parse a PE file, disassemble the code section, and
    communicate
# with ICE over a socket.
#

import distorm3
import pefile
import peutils

import argparse
import socket
import json
import os

# ----- Global Variables ----- #

gExecutableFileName = ''

COMM_HEADER_LEN = 4
COMM_HOST = "localhost"
COMM_PORT = 4040

# ----- Identifying & Creating Functions ----- #
```

```
class Function (object):
    def __init__(self, startEA, endEA, startSlice, endSlice):
        self.startEA = startEA
        self.endEA = endEA

        self.startSlice = startSlice
        self.endSlice = endSlice

        self.name = "sub_"
        hs = hex(self.startEA)[2:]
        if hs[-1] is 'L':
            self.name += hs[:-1]
        else:
            self.name += hs

    def __str__(self):
        return "Function: (%s) [%s : %d, %s : %d]" % (self.name,
           hex(self.startEA), self.startSlice, hex(self.endEA), self
           .endSlice)

def generateFunctions(calls, instructions):
    funcs = []

    for c in calls:
        if c.operands[0].type != distorm3.OPERAND_IMMEDIATE:
            continue

        startEA = c.operands[0].value
        startSlice = seekAddress(instructions, startEA)

        (endEA, endSlice) = findNextReturn(instructions, startSlice)

        if (startEA != None) and (endEA != None):
            funcs.append(Function(startEA, endEA, startSlice,
               endSlice))

    return funcs

# ----- Instruction Operations ----- #

def seekAddress(instructions, address):
```

```
    for seed, instr in enumerate(instructions):
        if instr.address == address:
            return seed

    return None

def findNextReturn(instructions, seed):
    if seed is None:
        return (None, None)

    for idx, i in enumerate(instructions[seed:]):
        if i.flowControl is "FC_RET":
            return (i.address, seed+idx)

    return (None, None)

# ----- Disassembly & PE Parsing ----- #

def parsePE(exe):
    pe = pefile.PE(exe)

    ib = pe.OPTIONAL_HEADER.ImageBase
    cb = pe.OPTIONAL_HEADER.BaseOfCode
    cs = pe.OPTIONAL_HEADER.SizeOfCode
    data = pe.get_memory_mapped_image()[cb:cb+cs]

    return (data, ib, cb)

def disassemble(exe):
    (data, ib, cb) = parsePE(exe)

    instructions = distorm3.Decompose(ib+cb, data, distorm3.
       Decode32Bits, distorm3.DF_NONE)

    return instructions

# ----- Communication With ICE ----- #

def buildMessage(_action, _data):
    _instance_id = os.getpid()
    _origin = gExecutableFileName
```

```
    return json.dumps({'instance_id': _instance_id, 'origin':
       _origin,
                       'action': _action, 'data': _data})

def handle_request_functions():
    pass

# @return string - json reponse
def handleICERequest(req_json):
    req = json.loads(req_json)
    if req is None:
        return None

    if req['action'] not 'request':
        return None

    if req['actionType'] is 'functions':
        return handle_request_functions()
    elif req['actionType'] is 'calls':
        print "Request (calls) not handled yet"
    elif req['actionType'] is 'updateCursor':
        print "Request (calls) not handled yet"
    elif req['actionType'] is 'setComment':
        print "Request (calls) not handled yet"
    elif req['actionType'] is 'rename':
        print "Request (calls) not handled yet"
    elif req['actionType'] is 'cfg':
        print "Request (calls) not handled yet"
    else:
        print "Unhandled request type: " + req['actionType']
        return None

# @return none
def sendToICE(ice_socket, req_json):
    req_len = len(req_json)
    if req_len > 0xFFFF:
        return

    hi = req_len & 0xFF00
    lo = req_len & 0x00FF

    data = chr(hi) + chr(lo) + "\r\n" + req_json
```

```
    ice_socket.send(data)

# @return string - json request
def recvFromICE(ice_socket):
    header = ice_socket.recv(COMM_HEADER_LEN)

    hi = header[0]
    lo = header[1]

    # header[2:3] are ignored since they are
    # the sequence "\r\n" and not needed

    msgLen = ((hi << 2) | lo)

    print "recFromICE msgLen: " + hex(msgLen)
    msg = ice_socket.recv(msgLen)

    return msg

def connectToICE():
    ice = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    ice.connect((COMM_HOST, COMM_PORT))

    hello = buildMessage('hello', None)
    sendToICE(ice, hello)

    while True:
        req = recvFromICE(ice)
        resp = handleICERequest(req)
        if resp != None:
            sendToICE(ice, resp)

# ----- Entry Point ----- #

def main():
    parser = argparse.ArgumentParser(description='Disassemble an
       executable and provide it to ICE.')
    parser.add_argument('executable', metavar='executable',
                        help='Executable to disassemble')
    args = parser.parse_args()
```

```
    global gExecutableFileName
    gExecutableFileName = os.path.basename(args.executable)

    instructions = disassemble(args.executable)

    # We need to do this filter ourselves because the Python
       interface
    # of distorm3 does not properly handle the DF_RETURN_FC_ONLY
       flag.
    # If you use that flag there will be a mismatch between the
       values
    # received in the Instruction.instructionBytes field and the
       address
    # that is being referred too.  See the DecomposeGenerator method
    # in the distorm3 Python API and the usage of instructionBytes/
       instruction_off
    # for more info.

    calls = [i for i in instructions if i.flowControl is "FC_CALL"]
    rets = [i for i in instructions if i.flowControl is "FC_RET"]
    sys = [i for i in instructions if i.flowControl is "FC_SYS"]
    unc_jmps = [i for i in instructions if i.flowControl is "
       FC_UNC_BRANCH"]
    cnd_jmps = [i for i in instructions if i.flowControl is "
       FC_CND_BRANCH"]
    ints = [i for i in instructions if i.flowControl is "FC_INT"]

    funcs = generateFunctions(calls, instructions)

    connectToICE()

if __name__ == "__main__":
    main()
```

Appendix B

Malware Analysis Tools

The following tools were used to perform the malware analysis of Section 3.2.7.

Kernel Based Virtual Machine (KVM)

KVM is a popular virtualization platform for Linux. It is used during the malware analysis to provide a means of isolating the malware from the network and to allow for easy restore of a clean virtual machine. More information about KVM can be found at *http://www.linux-kvm.org/*.

IDA Pro

IDA Pro is the industry-standard disassembler. It is capable of disassembling a wide range of binaries and provides many features to assist with reverse engineering. Hex Rays is the developer of IDA Pro and has information about it located at *https://www.hex-rays.com/products/ida/index.shtml*.

Import REConstructor

Import REConstructor (ImpREC) is a tool that allows you to rebuild the Import Address Table (IAT) of an executable. Malware will often intentionally damage the IAT in order to obfuscate its behaviour during static analysis. ImpREC can be obtained from *http://tuts4you.com/download.php?view.415*.

INetSim

INetSim is used to simulate a typical network and analyze the traffic of unknown malware. It is run on a Linux host and is able to simulate various services typically found on a network such as HTTP/HTTPS, SMTP/SMTPS, DNS, among

many others. The INetSim project page is located at *http://www.inetsim.org/*.

LordPE

LordPE is a tool to dump an executable from memory and is capable of editing PE files. It was used to dump the original malware executable from memory once it has been unpacked. LordPE can be found at *http://www.woodmann .com/collaborative/tools/index.php/LordPE*.

OllyDbg

OllyDbg is a debugger for the Microsoft Windows platform. It serves as a way to step through the malware during execution and helps to evade obfuscation and anti-reverse engineering techniques employed by the malware. OllyDbg is a free download available at *http://www.ollydbg.de/*.

PEiD

PEiD analyzes an executable to identify what type of packer, if any, was used on it. It is capable of identifying a wide range of packers that are commonly used in malware. PEiD is available from *http://tuts4you.com/download.php?view.398*.

PEView

PEView is a tool to view structural aspects of a PE file. PEView allows for analysis of the PE file header and inspection of the segments contained within the file. PEView can be downloaded from *http://wjradburn.com/software/*.

Process Explorer

Process Explorer (Procexp) is a utility for Microsoft Windows that displays currently executing processes along with detailed information about each. Procexp is part of the Sysinternals package and is available from *http://www.sysinternals.com/*.

Process Monitor

Process Monitor (Procmon) is a utility for Microsoft Windows that tracks various operations performed by a process. It is capable of tracking operations such as forking threads and processes and accessing registry values as well as a large number of other operations. Procmon is part of the Sysinternals package and is available from *http://www.sysinternals.com/*.

Resource Hacker

Resource Hacker is static analysis utility to inspect, modify, and extract data from the resource section of a PE file. Resource Hacker is available online at *http://www.angusj.com/resourcehacker/*.

Strings

Strings is a utility that analyzes a binary for static strings and displays them to the user. Strings are extremely useful when performing malware analysis as they may provide information such as malicious URLs, text used for debugging, or other textual data that indicates possible malware behaviour. Strings is part of the Sysinternals package and is available from *http://www.sysinternals.com/*.

Wireshark

Wireshark is a popular packet capture and analysis program. Wireshark is capable of performing analysis on the packet stream and saving the raw packets to file for later analysis. It is used in malware analysis to inspect the network traffic of a malware sample and can be found on its project page at *http://www.wireshark.org/*.

VirusTotal

VirusTotal is an online utility that analyzes a given sample of malware and produces a report. The report includes detection information by numerous anti-virus solutions and information obtained from running several tools on the provided sample. VirusTotal is located at *https://www.virustotal.com/*.