Progressive User Interfaces for Regressive Analysis: Making Tracks with Large, Low-Level Systems

Jennifer Baldwin[1] **Prosenjit Sinha**[2] **Martin Salois**[3] **Yvonne Coady**[1]

[1] Department of Computer Science
University of Victoria
Victoria (BC) Canada
Emails: jbaldwin@cs.uvic.ca, ycoady@cs.uvic.ca

[2] Department of Computer Science
Concordia University
Montreal (QC) Canada
Email: p_sinh@encs.concordia.ca

[3] Defence Research and Development Canada (DRDC) Valcartier
Quebec City (QC) Canada
Email: martin.salois@drdc-rddc.gc.ca

Abstract

Comprehension of low-level issues, such as malware threats, often relies on dated user interfaces that actually inhibit navigation and exploration of large code bases. These user interfaces often fail to exploit visualization techniques that could significantly alleviate cognitive overhead. An initial usability survey reveals that better and easier analysis of control flow is particularly critical for malware program comprehension.

By developing tools that couple high-level views of control flow relationships with more detailed views of call sequences, we demonstrate how improved user interfaces can leverage visualization techniques. These tools go beyond the ubiquitous call graph and have the ability to scale in ways that promote their use for comprehending large, complex systems.

1 Introduction

Reverse engineering is complex and time-consuming, particularly when obfuscated in malware. The current lack of modern visualizations in assembly language tool support further exacerbates this problem. Whereas engineers of higher-level systems often rely on tools for effectively navigating codebases and analyzing design, corresponding support for lower-level systems is severely lacking.

For example, consider the ways in which a typical call diagram is presented to developers in a state of the art disassembler and debugger: IDA Pro (Hex-Rays SA 2010). Figure 1 shows a function call graph generated by executing the Mariposa bot (Sinha et al. 2010) and analyzing the memory dump.

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

To address some of these issues, the Assembly Visualization and Analysis (AVA) project was started in collaboration with CA. The main goal of this project is to design, implement and evaluate program comprehension tools for large, low-level monolithic systems written in assembly (mainframe IBM HLASM, x86). Preliminary work proposes a new user interface for these tools designed to reduce the cognitive overload inherent in malware comprehension (Baldwin et al. 2009). The contributions of this paper are:

1. a usability survey evaluating work flow and associated tools for malware comprehension
2. requirements derived from the survey for new user interface (UI) features required for assembly language comprehension
3. our own UI support in a tool called *Tracks*, which allows developers to interact effectively with a sequence diagram to follow the flow of control in sync with IDA Pro debugging or static analysis

It is our hope that Tracks will help reduce the cognitive overload associated with malware analysis and other software reverse engineering tasks. This will be user tested in the AVA project.

This paper is organized as follows. Section 2 discusses related work. Section 3 presents the survey results and Section 4 focuses on one tool that we have created: Tracks. Section 6 provides a case study for Tracks using the Mariposa bot. Section 7 gives an analysis of these visualization techniques along with current limitations of the Tracks prototype. The paper closes with future work and a conclusion.

2 Related Work

There has been little work in the area of control flow visualization for malware analysis and assembly in general. Most notably, Quist & Liebrock (2009) provides VERA (Visualization of Executables for Reversing and Analysis), a graph that uses basic blocks as

This work was funded by DRDC contract W7701-82702/001/QCA and by CA Labs. It is now funded by a DND/NSERC grant (DNDPJ 395197 - 09).

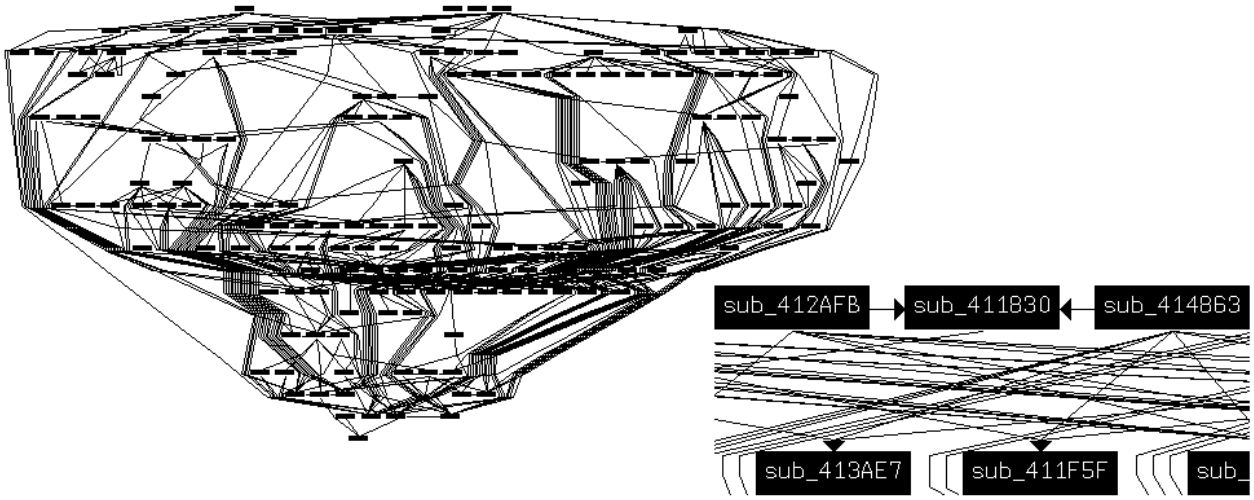

Figure 1: Mariposa function call graph from IDA Pro (with zoom-in)

nodes to support dynamic analysis. In most tools, the visualization is limited to a static function call graph and its associated static control flow graph for each function (Hex-Rays SA 2010, Zynamics 2010). There is no link with dynamic information. Other tools are limited to text interfaces with very limited access to modern development environment's UI features.

For languages other than assembly, (Bohnet & Döllner 2006) shows the value of visually exploring call graphs to find features in large C/C++ systems (over 1 million lines of code). Their approach is a control flow graph that combines dynamic and static analysis techniques. Our work differs from these two by focusing on sequence diagrams for assembly code.

One tool that is discussed in further detail in the design and implementation section is Diver (Bennett et al. 2007). Diver is an open source, extensible, sequence viewer that is capable of viewing control flow from various sources. Its user interface was designed to be scalable to huge control flow sequences and has been shown to be useful for large Java call graphs.

Other control flow tools exist for higher-level languages such as Code Bubbles (Bragdon et al. 2010), an IDE for Java allowing the creation of bubbles containing methods that are linked when the user selects a static call in the source code. They also provide bubbles for notes and status flags for easy documentation, a topic that came up often in our survey. But, for assembly, there is very little visualization support.

3 Malware Analysis Survey

To compile the most important problems for malware comprehension and what might be valuable opportunities for visualization, we surveyed fifteen malware analysts with different levels of experience.

The survey questions were an exploration of assembly code comprehension and work flow processes in general, combined with more specific questions. These specific questions were asked in response to several interviews that we did with analyzers at DRDC Valcartier and Concordia University, as well as with developers at CA Labs. Some of the questions were also based off of previous work such as the existence of beacons and important features of debugging. These are discussed in more detail below.

3.1 About the Respondents

The first section of the survey established the characteristics of the developers. Table 1 outlines the profiles of our respondents[1].

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

Table 1: About the respondents

3.2 Assembly Experience

Table 2 summarizes the results for assembly experience, where SD is the standard deviation. Respondents reported being slightly less adept at writing assembly then they were at understanding it, averaging around 2.9 and 3.5 respectively on a 1 - 5 Likert scale (Likert 1932). 80% of respondents thought that assembly is more difficult to understand than other languages. We also asked if there were more difficult languages than assembly, what their most familiar assembly dialect was and how they worked with assembly.

They were also asked what the most difficult task they had to perform was as well as which took the longest. The top reported most difficult task and time-consuming task was *following data and control flow*. The tasks that were the most time consuming were *trying to locate a certain behavior within the code*, *control flow analysis*, *data flow analysis*, *deobfuscation* and *decryption*.

3.3 Current Tools

The tool primarily used for malware analysis was IDA Pro at 87% (13/15). Table 3 outlines the tools used and their strengths and weaknesses. The most reported deficiency with current tools was that there

[1] Users could list more than one in these categories.

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

Table 2: Assembly experience

is no integration between them, no way to link their best features. This is an issue we hope to address within AVA.

It is important to note that the industry proposes other good tools for malware analysis such as Norman's Sandbox Analyzer Pro (Norman ASA 2010), Sunbelt's CWSandbox (Sunbelt Software, Inc. 2010) and HBGary's Responder (HBGary 2010). These tools, however, are usually extremely expensive (in the tens of thousands of dollars (USD)) and were out of reach for our respondents.

3.4 Browsing and Navigation

When trying to form a bird's eye view of the system, it is important to provide varying degrees of granularity. For this reason, we targeted questions as to what beacons exist in assembly. A *beacon* is a recognizable, familiar feature in the code that acts as a cue to the presence of certain structures. Beacons are used to move from high-level abstractions or concepts to lower-level details (Brooks 1983). Table 4 summarizes the answers for beacons and other aspects that could be used for navigation, thereby reducing cognitive overhead. Coding conventions include load/store, used to make a function call for example.

3.5 Debugging

We wanted to compare existing theories of comprehension about debugging with assembly (Erdös & Sneed 1998). Table 5 shows how important such debugging features were, on a Likert scale of 1 - 5.

These results show that the top two most important features pertain to control flow, further highlighting its importance in this domain. One area of concern in a few sections of the survey is the ability to change input values and run the system again.

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

Table 3: Current tool use

Asked	Reported
Beacons	27% (4/15) Function Calls (Control Flow) 27% (4/15) Data Usage 27% (4/15) Coding Conventions 20% (3/15) Function Definitions
Task-focused interface	100% (6/6) Yes
Zoom	33% (5/15) Functions 20% (3/15) Modules

Table 4: Browsing and navigation

3.6 Control Flow

The first question about control flow asked about static control flow scenarios. Two scenarios were used in the questions. The first is a static control flow that shows all of the functions that could possibly be called from a current function. The second is a history view which diagrams the functions the user has navigated to within IDA Pro. We asked if the developers could see any other static scenarios. They did not list any but did show concern for how loops, recursion and random locations would be handled.

We also wanted to know how useful reversed control flow would be. Reversed control flow means that given a function, one can step backwards to see what paths led there. We asked how useful each was on a Likert scale of 1 - 7. We use a 7-point scale for greater accuracy on this question because it is not asked in

Feature	$\overline{\chi}$	SD
Where is a particular subroutine/procedure invoked?	4.80	0.41
How does control flow reach a particular location?	4.60	0.51
Where is a particular variable set, used or queried?	4.60	0.63
What are the arguments and results of a function?	4.47	0.83
Where is a particular data object accessed?	4.33	0.82
What are the inputs and outputs of a module?	4.13	0.92
Where is a particular variable declared?	3.53	1.51

Table 5: Debugging features

Asked	Reported
Static concerns	20% (3/15) Loops and recursion
Dynamic concerns	7% (1/15) Multi-threaded traces 7% (1/15) Compare traces 7% (1/15) Branch frequency
Forward control flow	6.40 (out of 7) 0.87 SD
Reversed control flow	6.15 (out of 7) 1.07 SD
Reversed flow useful	87% (13/15) Yes
Information to mine	47% (7/15) System call patterns 13% (2/15) Compare traces 12% (2/15) How to reach Execution Point (jump conditions)

Table 6: User requirements for control flow

other ways elsewhere. Forward control flow was rated at 6.4 and reversed at 6.15. We also asked "yes or no, is reversed flow useful?" 87% said that it was.

Next we looked at dynamic control flow. We asked if the developers could see any other dynamic scenarios other than an execution trace. They did not but were interested in how it would deal with multi-threaded traces. That is being able to compare traces and seeing how often a branch is taken.

Finally we asked what information developers might want to mine from the control flow data. Table 6 summarizes these results.

3.7 UML Diagrams

Since sequence diagrams have shown promise for control flow, as discussed in this paper, an interesting question was whether or not there are other UML diagrams that might also be useful. 33% (5 out of 15) of the developers felt that state diagrams would be useful. 13% (2 out of 15) thought that activity diagrams would be useful. A lone response suggested package diagrams, with packages representing modules.

3.8 Requirements Summary

Table 7 summarizes all of the previous requirements for malware analysis.

Although this paper focuses on the control flow aspect of the requirements, we hope to address multiple areas of user-interface design in the AVA project (e.g. better integration, code base browsing and navigation, better debugging support, state diagrams).

4 Tracks: Interfacing with Control Flow

One of the most difficult challenges identified in our survey is to follow the control flow in assembly due to the inherently unstructured nature of assembly code. Thus, our first step in developing a modern user interface was to improve visualization and interaction for call graphs.

To address this issue, Tracks was built using Diver (Myers 2010), an open source tool developed by the Chisel Group at the University of Victoria and is discussed in more detail in Subsection 5.1.

Diver is primarily designed to show Java code and provides features for extremely large traces. For example, users may set any of the functions as the root of the diagram to reduce the amount of information displayed. There are breadcrumbs at the top of the diagram to navigate back to the previous view. There is also an outline pane to quickly navigate around the diagram and the state of the diagram can be saved.

Asked	Reported
Existing tool support	Better integration Instruction definitions Documentation support Higher-level representation
Browsing and navigation	Function calls Coding conventions Data usage Function definitions Modules
Task-focused	Documentation (Notes, bookmarks, tagging)
Debugging	Save debugging state Reset values/rerun Data flow Static simulation Multi-threaded support for IDA Pro
Control flow	System call patterns Loops and recursion Compare traces Data to reach execution point Branch frequency Multi-threaded traces
UML diagrams	State diagrams Activity diagrams Package diagrams

Table 7: User requirements for malware analysis

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

A sample sequence diagram is shown in Figure 2. At the top of this figure, a panel shows which file a function is defined in. Here we can see that `LocalFree` comes from `KERNEL32.dll`. When an imported function is selected, its control flow data, if it exists, is parsed and added to the diagram.

There are three separate views for call graphs: static and dynamic control flows, and navigation. We now explain each of these views in further detail.

4.1 Static Control Flow

In order to visualize static control flow, we need data for all the function calls. This call data was retrieved using a custom IDA Pro plugin iterating through each address of each function, checking for cross references and external calls.

Tracks reads this data and shows a tree containing all of the functions defined within a module. Selecting a function displays its static call graph, as shown in Figure 2. The user can then expand the function calls of interest. Functions with an `I` icon are functions imported from another file (e.g. a library).

4.2 Dynamic Control Flow

This view is synchronized with IDA Pro such that it receives messages whenever a new function is executed during a debugging session. If the diagram is opened from a function in the tree view, a breakpoint is set at that function in the target program (optional). The user has two choices for this diagram: to diagram all of the calls (enables step tracing within IDA Pro), or to diagram just the calls that are

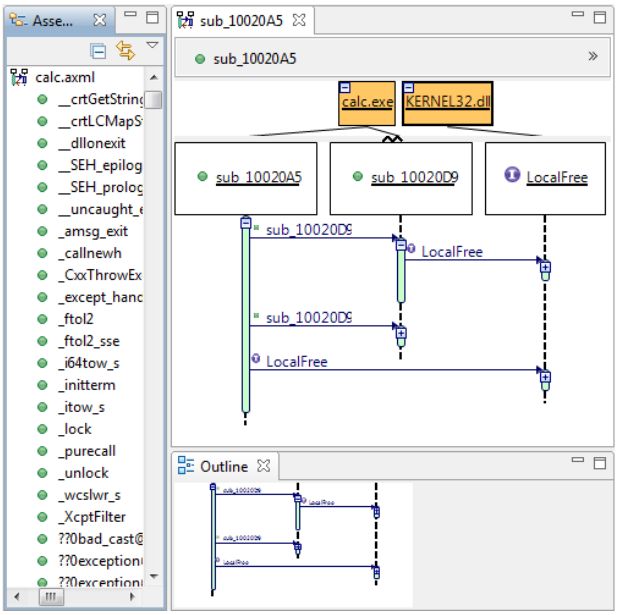

Figure 2: Tracks for the calculator executable

stepped into. When diagramming only the calls that are stepped into, hitting a breakpoint adds that function call to the *User* lifeline. There is also an option to trace calls within imported modules.

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

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

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

4.3 Navigation History

This diagram is similar to the dynamic control flow diagram in that it is built dynamically and uses the *User*'s lifeline as the root. The difference is that function calls are added to the diagram as the user navigates through IDA Pro. Selecting a function in IDA Pro adds it as a call from the *User*'s lifeline. Then, selecting a cross reference (or call) from within that function adds it as a call from that function's lifeline. Conversely, selecting a function that is not a cross reference adds it as a call from *User*.

This feature can be used, for example, to understand a particular path in the program and to document this understanding.

5 Tracks: Design and Implementation

This section discusses the implementation of the Tracks sequence diagram tool.

5.1 Diver: The Sequence Explorer

To create Tracks, we extended Diver, an open-source and extensible sequence diagram tool built using the Eclipse framework (The Eclipse Foundation 2010*a*). The design of Diver has two primary goals: model independence and interactivity/navigability.

Model independence means that the viewer is not tied to any particular model or data format in its back-end. The viewer has been employed to visualize program control flow from various sources, such as control flow of assembly language (in this research), dynamic traces from instrumented Java programs (Bennett et al. 2008) and the call structures of static Java source code. This has been accomplished using a framework compatible with the Eclipse JFace viewer framework (The Eclipse Foundation 2010*b*).

The second goal of interactivity and navigability was inspired by the fact that sequence diagrams can quickly become very large and extremely complex. The viewer has features to help overcome this problem. A short listing of the features includes: animated layout, highlighting of selected elements and related sub-calls, grouping of related calls (such as loops), hiding/collapsing of call trees and package or module structures, customizable colors and labels for visual elements such as activation boxes and messages, keyboard navigation through components and the ability to focus the sequence diagram on different subparts of the call structure. These features were studied and evaluated by Bennett et al. (2008).

5.2 Tracks Extension

In order to create the Tracks extension, we first defined XML models that contained all of the static and dynamic control flow information extracted by our IDA Pro plugin. Next, we defined our own content and label providers for the sequence diagrams so that they could be read and built in the diagram. For navigation history and dynamic diagrams, we build the diagrams dynamically and save to a different XML model because in this case, a function makes different calls at different times. We also added additional functionality such as the function tree view, saving diagram state, detecting cycles within the diagram, renaming functions, showing external calls and adding custom functionality for events such as navigation and setting breakpoints.

Note here that Tracks supports multiple IDA Pro instances. This was necessary because IDA Pro is single-threaded and we may need to debug more than one executable at a time (e.g. a program and its libraries). We will also see later that Mariposa, for example, injects code into `explorer.exe` which then needs to be debugged separately in another IDA Pro instance. This means that Tracks can show a complete dynamic control flow graph, but it also means that if you double click on an element, Tracks navigates to the correct IDA Pro instance.

5.3 Integration with IDA Pro

To pass messages between IDA Pro and Tracks, we needed to create an IDA Pro plugin that was able to listen to events and generate the required data as well as perform actions. We send these messages using sockets since IDA Pro plugins are written in C++ and Tracks is written in Java.

Figure 3 shows a sample of messages that can be exchanged between the two applications. The first message from Tracks is to initiate contact with IDA Pro, which causes it to send back the path to the XML file describing the static control flow. Next the Tracks framework can receive navigation, debugging

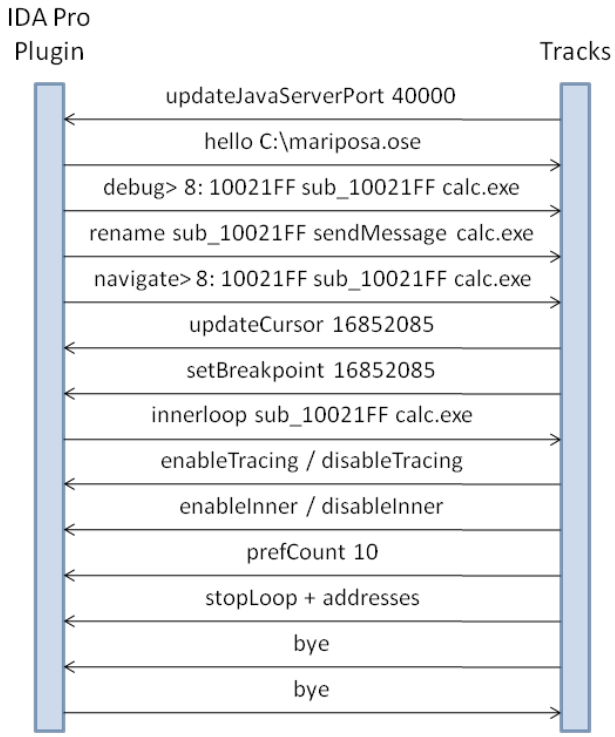

Figure 3: Messages between IDA Pro and Tracks

```
1  loc_13FFA6:
2      xor     dword ptr [ecx], 0CA1A51E5h
3      nop
4      add     ecx, 4
5      nop
6      nop
7      cmp     ecx, offset dword_41D4C0
8      jl      short loc_13FFA6
9      nop
10     push    offset loc_41D047
11     retn
```

Listing 1: Decryption loop in assembly code

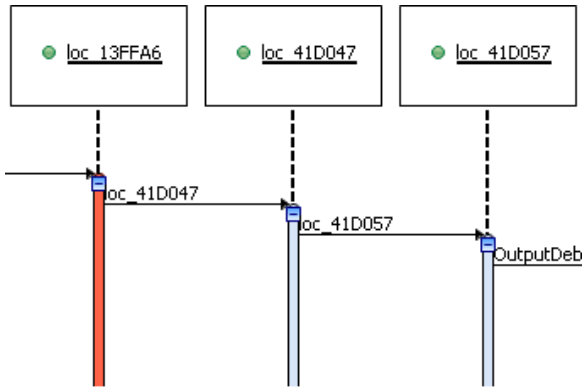

Figure 4: Decryption loop in the sequence viewer

and renaming events from IDA Pro, which contain additional information about functions. This information includes the index of the call (8 in this case), the function's address, the function's name and the file being debugged (in this case, `calc.exe`). The executable file name is needed when there are multiple IDA Pro instances in order to update the display in the right instance of IDA Pro. This message can also include the name of the external file where the function resides. Additionally, we can send events to all IDA Pro instances such as enable/disable tracing messages, enable/disable tracing calls within a library module, update preference count for loops, disable tracing of a specific loop and send goodbye messages.

6 Case Study: Tracks and Mariposa

Mariposa is a botnet, a collection of computers under the control of a single malicious entity. The most dangerous capability of this botnet is that it can download and execute arbitrary programs, which means the bot master can infinitely extend the functionality of the malicious software. This also reduces or eliminates the detection rates of traditional host detection methods. Due to this capability, 1500 variants of Mariposa have been detected so far and an estimated 12.7 million computer systems have been compromised (circa October 2009). We used information from Thompson (2010) and Sinha et al. (2010) to create the following three use cases, which show meaningful interactions through sequence diagrams.

The bot has three phases: obfuscation, decryption and injection. The obfuscation phase hides the intended functionality of the code. The decryption phase decrypts code and data that is used in the injection phase. The final phase, injection, is where Mariposa injects code into a legitimate system process in order to compromise the operating system.

We now look at some of these particular interactions in detail and show how the Tracks sequence diagrams can aid in identifying these interaction patterns. We use the Mariposa variant with the MD5 hash `3E3F7D8873985DE888CE320092ED99C5`.

There are two areas of interest: the detection of loops/cycles and the patterns of system calls as a means to detect malicious intent.

6.1 Obfuscation Phase

Obfuscation is the art of transforming *the application into one that is functionally identical to the original but which is much more difficult... to understand* (Collberg et al. 1997). Commonplace within malware, the Mariposa bot includes one such obfuscation, a large cycle that does nothing but useless computations, 889,976,605 times! The purpose is to confuse the person who is debugging the software and also, possibly, to confuse automated unpacking tools or malware analyzers. This obfuscation shows up as a cycle in Tracks, discussed in more detail in the injection phase (Subsection 6.3). After this loop, control flow is transferred to an address pushed onto the stack, which begins the decryption phase.

6.2 Decryption Phase

After the obfuscation phase, Mariposa must decrypt its own actual payload code.

The first decryption layer XORs[2] the range of data between addresses `0x41D000` and `0x41D4C0` with the constant `0x0CA1A51E5`. The address `0x41D047` is then pushed to the stack as a return value to transfer control flow to this address at function return. The assembly code for this first decryption layer is shown in Listing 1 and the corresponding sequence diagram is shown in Figure 4. Remember that large loops occurring within a single function are colored red to show that a large loop occurs, which may be suspicious.

6.3 Injection Phase

There has been much work in the security field to detect intrusion through sequences of system calls

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

```
1  loc_13591F:
2      lea     ecx, [ebp+var_128]
3      push    ecx
4      mov     edx, [ebp+var_134]
5      push    edx
6      mov     eax, [ebp+var_12C]
7      mov     ecx, [eax+6Ch]
8      call    ecx
9      test    eax, eax
10     jnz     loc_135899
```

Listing 2: Assembly code for finding each process

(Hofmeyr et al. 1998, Shankarapani et al. 2010). The idea being that short sequences of system calls executed by running processes can be a good discriminator between what is a normal process and what is an abnormal one. For example, finding code that modifies the registry that is not an installer or code that injects bytecode into another process could indicate malicious activity. Such code injection is a technique to hide malicious processes within a legitimate process and is a popular method for compromising an operating system. We hope that this type of information can be discovered more easily through the sequence viewer, at first manually by the user but automatically in the future.

In this section we look at system call patterns involved in Mariposa's injection process and their functionality once infected.

6.3.1 Preparing for Injection

The first step of injection is to prepare the data that is used by the injected code. This includes the creation of directories, getting the operating system version (need to check if `CreateRemoteThread` can be called) and creating files.

The next step is to find the process to inject into. To do so, `CreateToolhelp32Snapshot` is used to retrieve a snapshot of the processes that are running in the system. `Process32First` is called to retrieve the first of these processes and then `Process32Next` is called until the required process name is found. There are 107 lines of code total lines of code for this step, which cannot be shown due to space constraints. Listing 2 shows the code for the function that calls `Process32Next`. As we can see, the call is made on Line 8. However, since the address is stored in the register, it is not easily apparent statically that this is a call to `Process32Next`. This information becomes apparent while debugging and stepping through the call. It is even more apparent in the sequence diagram, as shown in Figure 5.

6.3.2 Injection

Having found a process to inject, the program calls `OpenProcess` to get a handle to this process. It then calls `VirtualAllocEx` to allocate memory within the target process, `NtWriteVirtualMemory` to inject the code and `CreateRemoteThread` to execute it. A diagram is not shown here due to space constraints.

6.3.3 Injected Process

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

Several interesting things occur within the injected process, as seen in Figure 6. These include ensuring that the files that will reinfect the system on startup are present and that their values are set in the registry and communication with the server is set. Here we only discuss this last step.

The infected process connects to the command and control server. The events that take place here are creating a socket with `32_ioctlsocket` and then `32_inet_addr` converts the domain names into proper addresses. Next, the bot retrieves host information from the host name by calling `32_gethostbyname`. The `32_htons` function converts an unsigned short number from the host to a TCP/IP network byte order (big endian). In order to authenticate with the server, an encrypted magic word is sent. Figure 6 shows a collapsed loop at this point in the diagram. This loop is responsible for encrypting the magic word. Once the magic word is encrypted, it is sent using the `32_sendto` function. It then receives a reply from the server using the `32_recvfrom` function (not shown). The injected process is then ready to decrypt and decode received commands.

7 Analysis: Is Value Added by Tracks?

Table 8 compares the ways in which IDA Pro and Tracks address the requirements from our survey. Relative to IDA Pro, it is our belief that the visualization in Tracks reduces cognitive overhead by better supporting navigation, showing a more intuitive control flow and allowing zoom/collapse interaction with visual cues. Additionally, added features integrated into this visual framework—system call patterns, dynamic traces, loops and recursion, call ordering and traces involving more than one executable— advance the state of the art without increasing cognitive overhead. The data is available to compare traces and analyze branch frequency in Tracks, whereas no data is easily available for export directly from IDA Pro.

VERA (Quist & Liebrock 2009) provides a high-level dynamic view of basic blocks, loops and color coding to describe where the code is located. It also provides some navigation links to IDA Pro. It currently provides a high-level view of the code to quickly pinpoint areas of interest, but it is not really useful after this point. The user is back in IDA Pro to understand the finer details. In contrast to VERA, Tracks provides three separate views of control flow in the form of conceptual sequence diagrams that support function names and calls as well as system calls. We believe these three views can be quite useful to speed up comprehension and document that understanding.

Code Bubbles (Bragdon et al. 2010) includes static call graphs, navigation support and also a debugger. However, while many of the features of Code Bubbles are useful, its focus is more on the code within the bubbles. There is no view that contains just the calls. It also does not focus on extremely large traces which we must contend with in assembly code.

For these reasons, we believe that Tracks can be quite useful and we will try to demonstrate this through user studies in future work.

7.1 Current Limitations

Important limitations remain in regards to using Tracks with IDA Pro, most notably, anti-debugging traps. Anti-debugging traps are used to detect if a program is running under the control of a debugger and to prevent this runtime debugging. Ideally, one would be able to run the executable with the sequence viewer open and afterwards investigate the entire call graph. Unfortunately with the anti-debugging traps

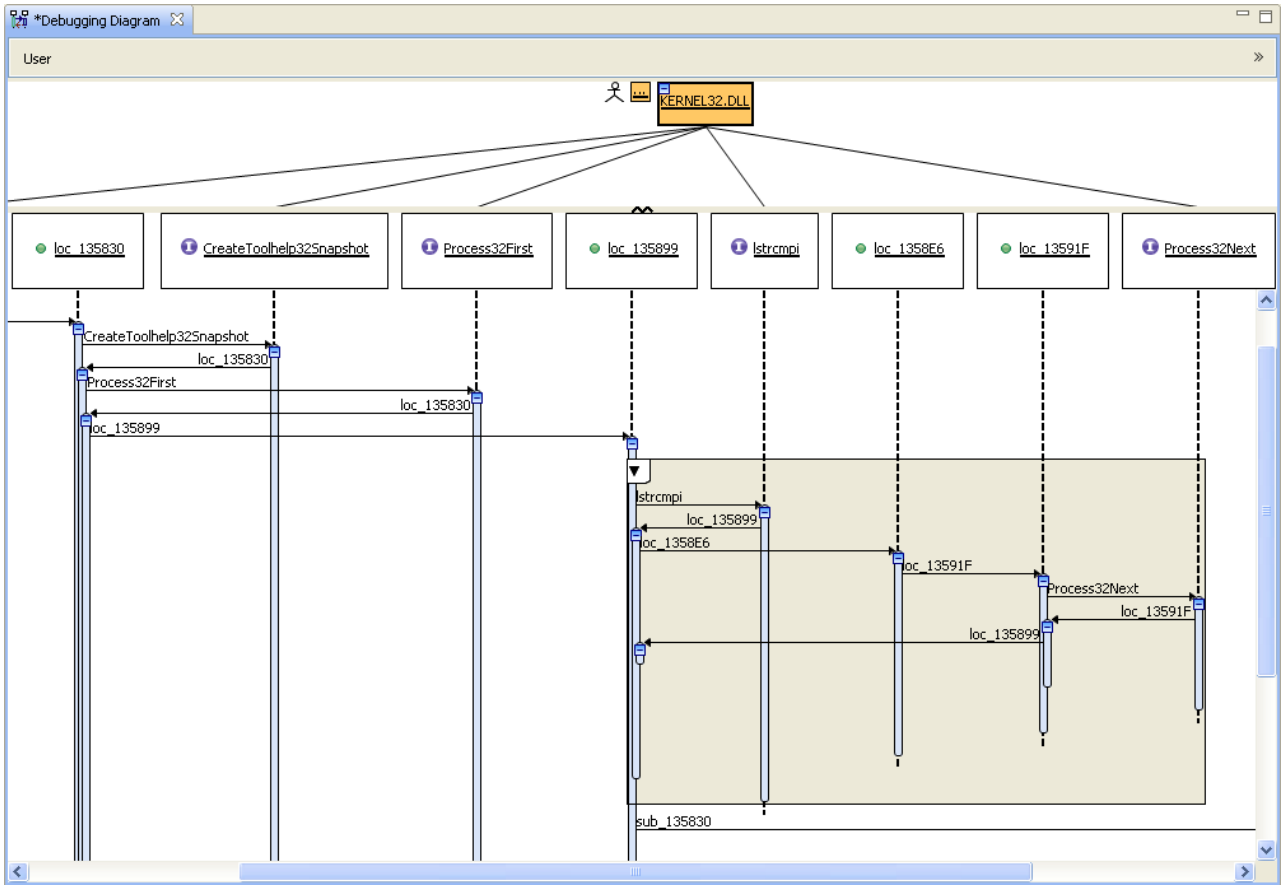

Figure 5: Finding the process to inject

within Mariposa, this is an impossibility, since we cannot single step over the code. Therefore it is up to the user to know how to debug the executable. This means that they have to have some prior knowledge about the system and cannot gain this prior knowledge directly from the sequence diagrams. We hope that better anti-anti-debugging tools will solve some of these issues. VERA addresses this through the Ether hypervisor framework, which is an avenue that could be investigated.

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

8 Future Work

There are many future avenues for future work in user interface strategies for Tracks and in related issues for comprehension. We first discuss what changes will be made in regards to Tracks, followed by other tools that we plan to implement and plans for user studies.

8.1 Sequence Viewer

From our survey, we can identify six areas of future work: 1- recognizing system call patterns, 2- creating documentation, 3- showing reversed control flow, 4- comparing traces, 5- finding the data required to reach execution points and 6- calculating branch frequency.

The system call patterns can be a good indicator as to whether or not a program is malicious. For example, we could mark a cycle as *Decryption*. Like Code Bubbles, we feel that it is important, when faced with so much information, to be able to make notes and flag items from within the tool. This is something we would like to contend with both in Tracks and future tools. Being able to provide a reversed control flow path was of great interest to show how to potentially reach a particular function. Comparing two traces to see how the program executes differently from one run to the next is also very important as well. It can show which data to use to execute a particular scenario. Lastly, branch frequency would indicate how often some code is run. This can be helpful to locate performance bottlenecks.

8.2 Further Tools

We discussed the need to reset the debugging state. The way analysts figure out how to run malware is an interative and tedious process, due to the difficulty in running past anti-debugging traps to specific points in the program execution. Even with all of the traps known, one has to be extremely careful re-running the system. A state diagram that allows the user to run the system automatically to a state such as `Injection` would save a lot of time and frustration.

Additional tools we decide to implement in the AVA project will support both the needs defined in our survey as well as those defined by the mainframe developer community, regardless of the underlying brand of assembly.

8.3 User Studies

An ongoing area of future work will be user studies. After our initial survey, we will recruit participants for user studies at multiple points during the research. It is important to see how users work with current tools in order to realize what the pain points are. We also

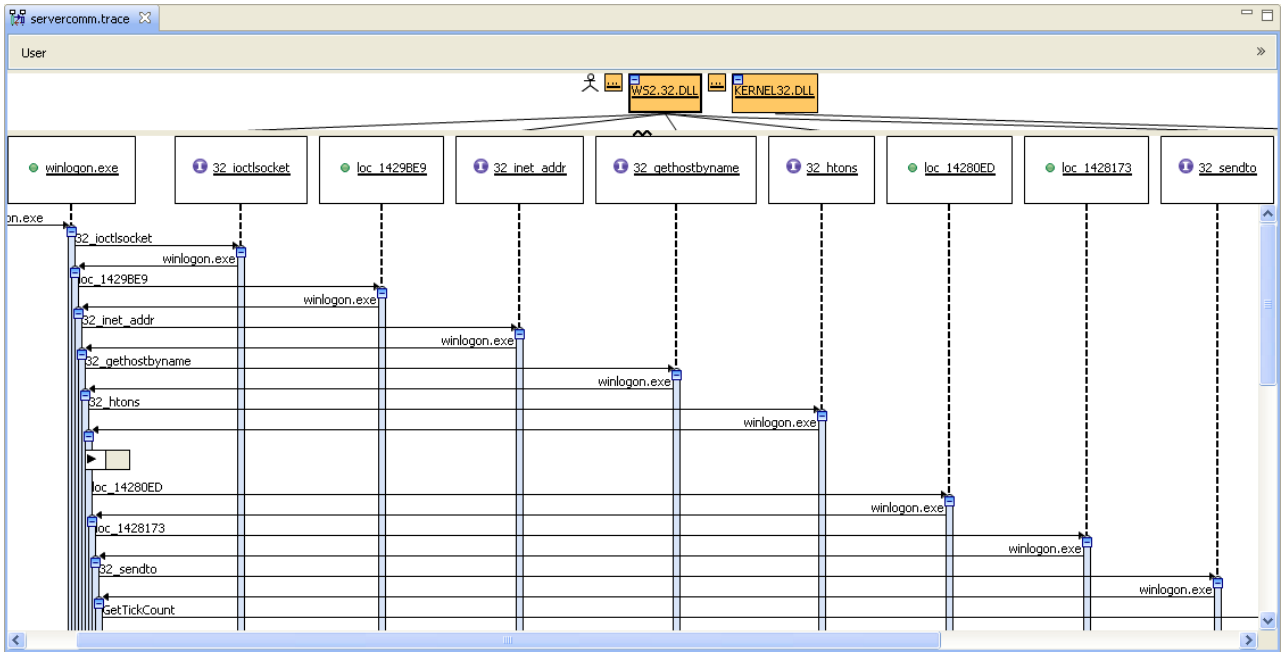

Figure 6: Communication with the server

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

Table 8: Comparison of IDA Pro and Tracks control flow tools

need to know whether or not the tools we build actually assist with the development and understanding of systems written in assembly. User studies will help determine how a work in progress, such as Tracks, will need to evolve as well as how it actually performs in comparison to the existing tools.

9 Conclusions

One of the goals of the AVA project is to develop tools and user interfaces that will assist reverse engineers working in cyber security. In order to best establish requirements for these user interfaces, we have performed a survey covering a wide range of issues involving assembly language comprehension. Partially in response to that survey, we have developed Tracks, which leverages progressive user interface techniques to improve support for control flow analysis. Tracks introduces additional features including following dynamic tracing in a sequence diagram, providing navigational aids, loop and cycle detection and integrating with IDA Pro.

Our analysis reveals ways in which work in the cyber security community can be enhanced through the application of visualization techniques and interactive user interfaces applied to low-level systems. Tracks is the first of several tools we plan to develop, attempting to bridge the gap between high-level tool support and low-level analysis tasks.

Acknowledgments

We would like to thank for their support Dr. Mourad Debbabi, from the Computer Security Lab at Concordia University, Del Myers and Margaret-Anne Storey, from the Chisel Group at the University of Victoria and David Ouellet, from DRDC Valcartier.

References

Baldwin, J., Myers, D., Storey, M.-A. & Coady, Y. (2009), Assembly Visualization and Analysis. An Old Dog CAN Learn New Tricks!, *in* 'Proceedings of the 2009 OOPSLA Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)', Orlando, FL.

Bennett, C., Myers, D., Storey, M.-A. & German, D. (2007), Working with 'monster' traces: Building a scalable, usable, sequence viewer., *in* 'In Pro-

ceedings of the 3rd International Workshop on Program Comprehension Through Dynamic Analysis (PCODA)', Vancouver, Canada, pp. 1–5.

Bennett, C., Myers, D., Storey, M.-A., German, D., Ouellet, D., Salois, M. & Charland, P. (2008), 'A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams', *Journal of Software Maintenance and Evolution: Research and Practice* **20**(4).

Bohnet, J. & Döllner, J. (2006), Visual exploration of function call graphs for feature location in complex software systems, *in* 'SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization', ACM, Brighton, United Kingdom, pp. 95–104.

Bragdon, A., Reiss, S. P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. & LaViola, Jr., J. J. (2010), Code bubbles: rethinking the user interface paradigm of integrated development environments, *in* 'ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering', ACM, New York, NY, USA, pp. 455–464.

Brooks, R. (1983), Towards a theory of the comprehension of computer programs., *in* 'International Journal of Man-Machine Studies', Vol. 18, pp. 534–554.

Collberg, C., Thomborson, C. & Low, D. (1997), A Taxonomy of Obfuscating Transformations, Technical Report 148, Department of Computer Sciences, The University of Auckland, New Zealand.
URL: `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a`

Erdös, K. & Sneed, H. M. (1998), Partial comprehension of complex programs (enough to perform maintenance), *in* 'IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension', IEEE Computer Society, Washington, DC, USA, p. 98.

HBGary (2010), Responder Pro.
URL: `https://www.hbgary.com/products-services/responder-pro`

Hex-Rays SA (2010), IDA Pro Disassembler.
URL: `http://www.hex-rays.com/idapro`

Hofmeyr, S. A., Forrest, S. & Somayaji, A. (1998), 'Intrusion detection using sequences of system calls', *J. Comput. Secur.* **6**(3), 151–180.

Likert, R. (1932), 'A Technique for the Measurement of Attitudes', *Archives of Psychology* **22**(140), p. 55.

Myers, D. (2010), Diver: Dynamic Interactive Views for Reverse Engineering.
URL: `http://diver.sourceforge.net`

Norman ASA (2010), Sandbox Analyzer Pro.
URL: `http://www.norman.com/technology/en`

Quist, D. A. & Liebrock, L. M. (2009), Visualizing compiled executables for malware analysis, *in* 'The 6th International Symposium on Visualization for Cyber Security (VizSec)'.

Shankarapani, M. K., Ramamoorthy, S., Movva, R. S. & Mukkamala, S. (2010), 'Malware detection using assembly and API call sequences', *Journal in Computer Virology* .

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

Sunbelt Software, Inc. (2010), CWSandbox.
URL: `http://www.sunbeltsoftware.com/Malware-Research-Analysis-Tools/Sunbelt-CWSandbox`

The Eclipse Foundation (2010*a*), Eclipse.org Home. http://www.eclipse.org.
URL: `http://www.eclipse.org`

The Eclipse Foundation (2010*b*), JFace - Eclipsepedia.
URL: `http://wiki.eclipse.org/index.php/JFace`

Thompson, M. (2010), Mariposa botnet analysis, Technical report, Defence Intelligence.
URL: `http://defintel.com/docs/Mariposa_Analysis.pdf`

Zynamics (2010), Zynamics BinDiff/BinNavi.
URL: `http://www.zynamics.com`