Spring 5-1-2017

Binary Analysis Framework

Josh Stroschein
Dakota State University

BINARY ANALYSIS FRAMEWORK

A graduate project submitted to Dakota State University in partial fulfillment of the requirements for the degree of

Doctor of Science

in

Cyber Security

May 2017

By
Josh Stroschein

Project Committee:

Dr. Wayne E. Pauli
Dr. Yong Wang
Dr. Matthew J. Miller

PROJECT APPROVAL FORM

We certify that we have read this project and that, in our opinion, it is satisfactory in scope and quality as a project for the degree of Doctor of Science in Cyber Security.

Student Name: Josh Stroschein

Dissertation Project Title: Binary Analysis Framework

Faculty supervisor:_____ Date: _____
Wayne E. Pauli, Ph. D.

Committee member:_____ Date: _____
Yong Wang, Ph. D.

Committee member:_____ Date: _____
Matthew J. Miller, Ph. D.

_____ Date:_____
Mark L. Hawkes
Dean for Graduate Studies and Research
Dakota State University

`

PROJECT APPROVAL FORM

We certify that we have read this project and that, in our opinion, it is satisfactory in scope and quality as a project for the degree of Doctor of Science in Cyber Security.

Student Name: Josh Stroschein

Dissertation Project Title: Binary Analysis Framework

Faculty supervisor: Wayne Pauli Date: 4/17/17
Wayne E. Pauli, Ph. D.

Committee member: Yong Date: 4/17/17
Yong Wang, Ph. D.

Committee member: Matt Miller Date: 4/17/17
Matthew J. Miller, Ph. D.

Mark Hawkes Date: 4-17-17
Mark L. Hawkes
Dean for Graduate Studies and Research
Dakota State University

ABSTRACT

The binary analysis of software has become an integral activity for security researchers and attackers alike. As the value of being able to exploit a vulnerability has increased, the need to discover, fix and prevent such vulnerabilities has never been greater. This paper proposes the Binary Analysis Framework, which is intended to be used by security researchers to query and analyze information about system and third party libraries. Researchers can use the tool to evaluate and discover unknown vulnerabilities in these libraries. Furthermore, the framework can be utilized to analyze mitigation techniques implemented by operating system and third-party vendors. The Binary Analysis Framework takes a novel approach to system-level security by introducing a framework that provides for binary analysis of libraries utilizing a relational data model for permanent storage of the binary instructions, as well as providing novel ways of searching and interacting with the parsed instructions.

Keywords: binary analysis, reverse engineering, vulnerability research

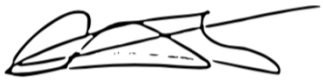
`

DECLARATION

I hereby certify that this project constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the project describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,



Josh Stroschein

`

TABLE OF CONTENTS

`

`

`

LIST OF TABLES

`

LIST OF FIGURES

`

`

CHAPTER 1

INTRODUCTION

This chapter introduces the reader to the current state-of-the-art in static binary analysis and binary analysis frameworks and shows the lack of a robust, multi-architecture, multi-platform framework for performing binary analysis. The chapter describes the creation of a framework for performing binary analysis and introduces a novel architecture for searching machine code to validate and discover software-based security features and vulnerabilities. The proposed framework will enhance security research by providing a multi-architecture, multi-platform binary analysis platform in a single framework. The framework will also provide a robust search architecture that utilizes disassembly output rather than machine code for search patterns.

This chapter provides the reader with an introduction to the problem and related background on topics pertinent to this work. Section 1.1 discusses the importance of performing static binary analysis and the security implications of undiscovered flaws in software design. Section 1.2 introduces the framework and the contributions it makes in the field of binary analysis. In section 1.3 the reader is given an overview of the current state-of-the-art in software security as well as how security features are bypassed by malicious actors. This section also covers prevalent attack techniques and fundamental architectural components of computing systems, concepts that are important to understand when discussing a framework designed to assist in security research. The final section, section 1.4, provides the layout of the remaining chapters of this work.

`

1.1 Introduction of the Problem

The daily use and interaction with electronic devices has become an integral part of everyday life. These devices are no longer only tools of commercial productivity, limited to the workplace, but have now permeated almost every aspect of modern life. Recent studies have found that adolescents aged 8 – 18 spend on average 7.5 hours a day on social media (i.e. electronic devices) (Drago, 2015). While studies such as this provide insight into the daily reliance of electronic devices, they often do not consider all the indirect interaction with electronic devices. From traditional devices such as laptops and desktops to devices where it is more difficult to observe the interaction, such as refrigerators, thermostats and vehicles, society is constantly interacting with and relying upon technology (*Technological Development and Dependency*, 2011).

However, the advancement of technology has not come without its perils. Massive data breaches, invasions of privacy, disruption to business operations and attacks against critical infrastructure are now greatly facilitated by the proliferation and inter-connected nature of these devices (Mandiant, 2016). This has given rise to a new profession, commonly referred to as "cyber security", in which professionals are tasked with providing the necessary insight to minimize the risk exposed by these devices, both commercially and at home (Francis & Ginsberg, 2016). Security engineer, network security architect, reverse engineer, malware analyst is a small sample of common titles given to those now on the front-lines of this new cyber age. Regardless of the title, one of the key tasks in evaluating the security of a device is in assessing the software running on that device. This software begins with the operating system, which is the first program loaded into memory when a computer starts and provides the required framework for other programs to run as well as other systems and users to interact with the computer ("Operating Systems," n.d.).

`

Attackers seek vulnerabilities, which are design weaknesses in the software running on a computer to gain unauthorized access (Dowd, McDonald, & Schuh, 2006). For an attacker to be able to compromise an operating system gives them complete control not only over that computer, but all the information the computer processes (Song et al., 2016).

In this paper, we propose the Binary Analysis Framework, a framework which will assist in performing reverse engineering activity on the operating system software to evaluate the code for known vulnerabilities and potentially discover unknown vulnerabilities. The Binary Analysis Framework will also extend to other software running within an operating system. With this information, security professionals can make significant contributions to securing our devices and minimizing the risks of the cyber-age.

1.2 Motivation

To evaluate the resilience of computing systems to malicious attack, it is crucial to be able to audit not only core operating system files but also third party software installed within the operating system. Therefore, the purpose of performing an audit of software is to not only identify systems susceptible to known vulnerabilities, but to also discover unknown vulnerabilities (Regalado et al., 2015). The auditing process provides owners of computing systems with the opportunity to update their software or apply other defenses to mitigate these vulnerabilities, reducing their exposure to attack.

1.2.1 Software auditing. The process of performing a software audit centers around the activity of reverse engineering, which is defined by Regalado et al. (2015) as "simply taking a product apart to understand how it works". The level of effort required to audit software will vary based on several conditions such as the complexity of the software, the scope of the audit,

`

availability of source code, technical proficiency of those performing the review and familiarity with the software under scrutiny. In the case where source code is not available, which occurs frequently with closed-source, proprietary software, binary analysis must be performed (Regalado et al., 2015). According to Andriesse, Chen, van der Veen, Slowinska, and Bos (2016), "Disassembly is thus crucial for analyzing or securing untrusted or proprietary binaries, where source code is simply not available". Binary analysis requires that the software being reviewed is done so from a binary state, without the benefits provided when analyzing source code such as variable and function naming, programmer comments, control flow constructs, and class definitions. This level of analysis requires greater knowledge by the reviewer in such areas as compiler behavior, operating system internals, assembly language, file formats and other topics that correspond to the inner-workings of the computer system (Regalado et al., 2015).

1.2.2 Binary Analysis Framework. Due to the complexities involved with binary analysis, this work proposes the Binary Analysis Framework, which is intended to be used by security researchers to query information about a variety of system and third party software. Security researchers can use the framework to audit software for known vulnerabilities as well as to perform research to potentially identify unknown vulnerabilities, providing an opportunity for remediation before the software is exploited. Furthermore, the framework can be utilized to analyze mitigation techniques implemented by operating system and third-party vendors, which assists in evaluating defensive measures in place on computing systems. The Binary Analysis Framework takes a novel approach to system-level security by introducing a framework that provides for binary analysis of software contained in the operating system as well as third-party software. In addition, it will contain information about exploit mitigation techniques that the software is, or is not, utilizing. These mitigation techniques are often enabled during compilation

`

or linking and not part of the original development of the software at the source-code level. This framework will utilize relational data models for permanent storage of the binary instructions as well as provide novel ways of searching and interacting with the parsed instructions.

1.3 Background

The advent of the digital computer was first proposed in a paper titled *On Computable Numbers* authored by Alan Turing ("History of Computing Hardware," n.d.). The advancement in digital computing hardware advanced drastically from the mid to late 20th century. The Electronic Numerical Integrator and Computer, or ENIAC, was the first electronic and programmable computer built in the United States. ENIAC weighed approximately 30 tons and required nearly 1800 ft^2 of physical space ("History of Computing Hardware," n.d.). By the beginning of the twenty-first century, cellphones had become a commodity item that were a fraction of the cost to manufacture, could fit in a person's pocket and contained approximately 1,300 times more computing power ("The ENIAC vs The Cell Phone," n.d.). The drastic reduction in size of computing hardware was equaled by the increase in performance, which is generally measured by a series of characteristics, such as system availability, resiliency, responsiveness, power consumption, cooling requirements and system throughput in terms of work capacity ("Computer Performance," n.d.). Indeed, modern cellphones are orders of magnitude more powerful than the computers used to land Apollo 11 on the moon (Zakas, 2013).

1.3.1 Commodity computing hardware. As the power of computing has increased over time (Schaller, 1997), there has also been a reduction in the cost of computing (Worthen, 2010), which was, in part, responsible for the rapid increase and availability of smaller, less expensive computing components. Computers such as the ENIAC, that were initially criticized as infeasible

`

were now proving their utility through performance as well as no longer suffering from cost and physical limitations, making them accessible to more businesses. The increased demand for computing hardware provided the incentive for the industry to continue to push for the development of faster, smaller and less expensive hardware (Reimer, 2005). This progression of smaller, less expensive hardware eventually gave rise to the possibility of the personal computer, or PC, which not only increased the use of computing hardware in businesses, but also created a market for personal use. The use of the personal computer ultimately created the potential for integrating computing technology in ways never conceived by the original inventors. Laptop computers introduced a mobile version of the personal computer ("History of Laptops," n.d.) and shortly thereafter the invention and rapid adoption of the smart phone ("Smartphone," n.d.). Fundamentally a smartphone does not differ significantly from a laptop, it is simply a personal computer running an operating system that is portable and able to connect to a cellular system (Hamblen, 2009, Mar 14). However, the smartphone does generally include features not commonly found in traditional personal computers such as Bluetooth capabilities, cameras, global positioning system (GPS) capabilities, and near field communication (NFC) capabilities, to name a few.

1.3.2 Internet of Things phenomenon. The integration of technology is continuing to expand into every day devices, from items as complex as airliners and automobiles down to those as simple as greeting cards. This expansion phenomenon is commonly referred to as the “Internet of Things (IoT)” (Morgan, 2014), and can be summarized as the process of providing computational and network communication abilities to common objects ("Hardware Control Flow Integrity (CFI) for an IT Ecosystem," 2015). A prime example that illustrates the “Internet of Things” phenomenon is the automobile, which has become an increasingly complex system of

`

computers and wireless communication interfaces (Wojdyla, 2013). One such system that is seeing advances in complex computing is in how drivers interface with the automobile. Automobile manufacturers are switching to steer-by-wire systems in which there is no longer a direct mechanical connection between the steering mechanism in the driver cabin and the wheels (Davies, 2014), computers provide the translation from physical inputs to mechanical instructions instead of mechanical systems providing this translation to other mechanical systems.

With the move to systems that rely on computation power instead of mechanical, there exists an increasing availability for vulnerabilities to exist. Dowd et al. (2006) defines a vulnerability as "flaws or oversights in a piece of software that allows attackers to do something malicious – expose or alter sensitive information, disrupt or destroy a system, or take control of a computer system or program." Vulnerabilities have been discovered in entertainment systems that have enabled an attacker to subvert control of the automobiles primary systems, namely acceleration, deceleration and engine operations (Huddleston, 2015). This highlights the importance that the underlying hardware and software have, especially as an increasing number of objects not only adopt computing power, but also the ability to communicate over the Internet.

1.3.3 Security implications. The wide spread adoption and usage of the Internet has provided a robust distribution network for the delivery of communications and software. While the clear majority of Internet-based activity is benign, there is a significant amount of activity that is related to malicious activity. According to Sikorski and Honig (2012), any software that "does something that causes harm to a user, computer, or network can be considered malware." It is estimated that malware costs approximately $13 billion annually to the world economy ("Malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code," 2007). In addition, it is estimated that the cost of a data breach, which is generally defined as the

`

act of stealing or taking information without the owner's consent ("Data Breach," 2016), now averages $3.8 million dollars, with an average cost per record of data at $154 ("Cost of data breaches increasing to average of $3.8 million, study says," 2015).

Other sectors of the economy may see a more significant financial impact from a breach such as the healthcare industry, which is estimated to have an average record cost of $363 ("Cost of data breaches increasing to average of $3.8 million, study says," 2015). Data breaches can often be traced to the introduction of malware, malware that often exploits vulnerabilities in operating system software or third-party code to achieve arbitrary code execution ("The Eight Most Common Causes of Data Breaches," 2013). That is, they can execute arbitrary code on the target platform, code that was never intended to be executed by the developers or even included in the original software design. This has created a back and forth in which malware authors, often referred to as attackers, are constantly looking for new and effective ways to distribute their malware while software vendors and security researchers are seeking ways to prevent attackers from exploiting computing systems. Once an attacker has been able to compromise a target system, they can not only harvest information processed on that machine, but also to expand their reach throughout the network. In the case of Target, who suffered a large data breach in 2013, attackers could leverage the compromise of a single host to ultimately install malicious software on a significant number of Target's point of sale terminals (Kassner, 2015) and compromise financial information for approximately 40 million of its customers (Krebs, 2014).

Auditing the attack surface of software, to include the operating system, becomes a crucial task for both security researchers and those charged with the security of computing devices. An operating system is defined as "system software that manages computer hardware and software resources and provides common services for computer programs. The operating system is a

`

component of the system software in a computer system. Application programs usually require an operating system to function" ("Operating Systems," n.d.). Put simply, an operating system is composed of software that manages the interactions between a user and the hardware. This software is the core system programs, services and libraries provided by the operating system developer. On the Microsoft Windows series of operating systems, these are often executable files or dynamically linked libraries (DLLs) ("What is a DLL?," n.d.). In addition, third party developers can also distribute their software through libraries and other executable programs to expand the functionality of the operating system and allow for a greater range of applications to run. A primary example of this is Adobe Flash, which provides for the installation of several libraries that are utilized by web browsers to execute application data that pertains to Flash applications, expanding the capabilities of modern web browsers.

When assessing the overall security posture of a system, both the operating system software as well as third party software must be considered (Manes, 2016). Data is collected about known vulnerabilities as they pertain to both operating system and third-party software by the National Institute of Standards and Technology (NIST) ("National Vulnerability Database," 2016). NIST provides a common approach to identifying, categorizing and even searching published vulnerabilities through the National Vulnerabilities Database (NVD). The format they use is provided by MITRE and called Common Vulnerabilities and Exposures, or CVE ("Common Vulnerabilities and Exposures," n.d.). The CVE format defines a standard for the classification and sharing of security information. The NVD contains over 75,000 related CVEs that pertain to operating system and third-party application software ("National Vulnerability Database," 2016). This data is used annually to produce a report highlighting the number of vulnerabilities (CVEs) by operating system and software (Manes, 2016). Figure 1 highlights the increasing trend of

`

known CVEs from 2015, which totaled 8,822. In 2014, the total number was 7,038 (Flores, 2015), that is an increase in 20% from the previous year and up nearly 60% since 2011, which had a total of 3,532 known vulnerabilities.

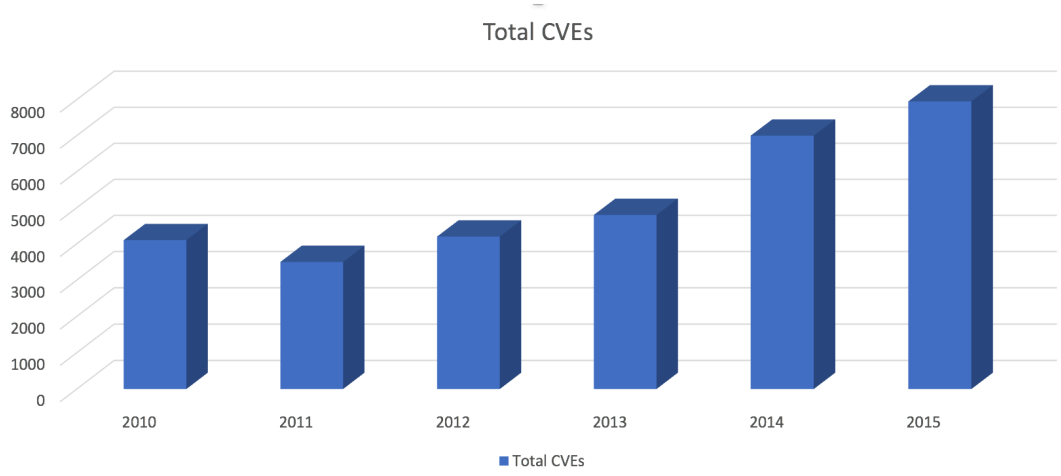


Figure 1. Total number of common vulnerabilities and exposures (CVEs) annually.

Data relating to known vulnerabilities enables us to divine trend information, which indicates that operating system and third-part software will continue to be exploited by attackers to gain unauthorized access.

Manes (2016) further breaks down each vulnerability type by vendor and operating system version. Table 1 highlights this data and represents the top 15 vendors. In the case of a vendor that produces multiple versions of their operating system, such as Microsoft, each version is listed individually. This data highlights that even though Microsoft did not have the most CVEs for a single version of its operating system, the aggregate of all versions means that most known vulnerabilities affected Windows. NetMarketShare produces real-time statistics of operating system market share by collecting information about operating systems through information transmitted commonly by web browsers and reports that the Windows operating system has 90%

of the market share, making it the most prevalent operating system in the market ("Operating system market share," 2016).

Table 1. Reported operating system CVEs.

Rank	Operating System	Number of Vulnerabilities
1	OS X	384
2	Microsoft Windows Server 2012	155
3	Ubuntu Linux	152
4	Microsoft Windows 8.1	151
5	Microsoft Windows Server 2008	149
6	Microsoft Windows 7	147
7	Microsoft Windows 8	146
8	Microsoft Windows Vista	135
9	openSUSE	121
10	Debian Linux	111
11	The Linux Kernel	77
12	Microsoft Windows 10	53
13	Fedora Linus	38
14	Microsoft Windows Server 2003	36
15	Xen OS	34

1.4 Common Memory Corruption Vulnerabilities

Many modern attacks that desire arbitrary execution of code involve the targeting of native software, such as operating systems, web servers and browsers. While not exclusive to these languages, programming languages that tend to expose memory corruption bugs are written in C or C++ (Ray, Posnett, Filkov, & Devanbu, 2014). This stems from the requirement in these languages for the programmer to handle the allocation, access (read/write) and deallocation of memory used by their programs. Mistakes in these tasks can then lead to exploitable vulnerabilities in the software. Juxtapose this with languages such as C#, which reduces this risk using an automatic memory management feature ("Automatic Memory Management," n.d.). That is, the resulting program is executed in a runtime which handles memory allocation, access and deallocation automatically for the program, eliminating the need of the programmer to handle these

`

tasks. The execution of a program in a runtime environment helps to reduce the impact of errors in memory management due to programming mistakes. However, even software written to run in a managed environment can expose memory corruption bugs as the runtime environment itself may be written in a language that executes in a non-managed environment. For example, Adobe Flash provides an execution environment for Flash applications. A Flash application is written in ActionScript before it is compiled into intermediary code, often referred to as bytecode. The Flash runtime environment is written in C++ ("Adobe Flash," n.d.) and is responsible for the execution of the Flash Application's bytecode, it translates the bytecode into specific machine instructions for the central processing unit (CPU) on the host system. Vulnerabilities that lead to the corruption of memory are commonly referred to as memory corruption, as the attack corrupts the process memory of the target application to subvert the normal flow of execution of the program and potentially achieve arbitrary code execution ("Memory Corruption," n.d.). This section will describe a few of the most prevalent memory corruption bugs.

1.4.1 Memory layout of a process. The execution of a program begins when the program is first loaded into memory. Normally, a program has been installed or is stored on a permanent medium, such as the hard drive on the host computer. However, program storage is not limited to a physical disk connected to the host computer and may extend to removable drives and even programs loaded directly into memory over a network. Once a program has been selected for execution, key components of the operating system must first parse the program data and load the program into memory. One of those key components is an operating system level utility called a loader and is responsible for parsing information about the program, selecting the environment in which it should be executed under, and then mapping the contents of the program into memory.

`

Once a program is loaded into memory, it is viewed on the system as a process, which acts as a container for all the program's resources. Each operating system can choose how to map a program into memory, but most follow some general conventions for the use of the memory space provided to the program. Figure 2 provides a high-level overview of a typical program that has been loaded into memory.

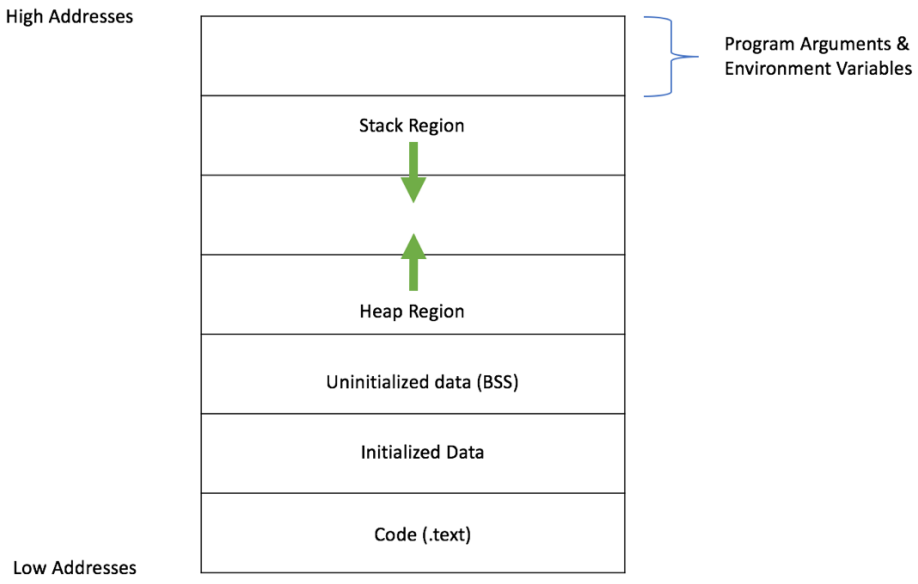


Figure 2. Representation of program layout in memory.

In Figure 2 common areas are highlighted and will be referred to as sections. Each section is given a range of addresses in which each address represents a byte of memory. The loader determines how much memory each section receives based off information that is parsed in the file format. Section size may also vary based off usage, such as the stack and heap. The executable instructions of a program are generally loaded into the code (.text) section. Data, both initialized and uninitialized, is loaded into the data and BSS sections, respectively (Erickson, 2008). The stack and heap regions of memory expand and contract based off program usage. Memory

`

corruption attacks occur when an attacker can change how or where the CPU receives its instructions. The following sub-sections describe prevailing memory corruption attack scenarios.

1.4.2 Stack buffer overflow. The program's stack stores both data and return addresses (code pointer) and is a critical abstract data structure responsible for storing information about current function execution as well as the calls leading up to the current function (Dowd et al., 2006). A stack buffer overflow occurs when a program using a buffer (i.e. allocated memory) in the stack region of memory can cause the program to write to more memory than was allocated for the buffer (One, 1996). A stack frame is created for each function call, this concept encompasses the arguments, return address and local variables (Sikorski & Honig, 2012). If the overflow can overwrite return information, an attacker can supply arbitrary data to subvert the execution of the program. Figure 3 highlights a stack frame layout for two functions. The stack frame on the right depicts the state of the stack when a buffer overflow occurs, while the stack frame on the left depicts normal program usage.

`

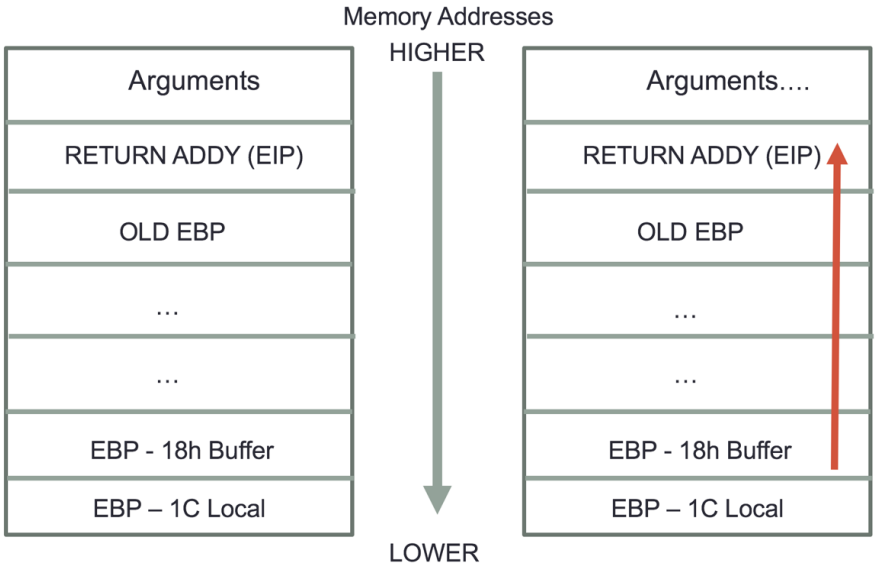


Figure 3. Stack frame for function vulnerable to buffer overflow.

In this scenario, the attacker can supply specially crafted data that overflows the buffer at *EBP-18h* and, by supplying enough data, overwrite a 4-byte value used to store the return addresses. When the program returns from the function is uses the corrupted value instead of the original and the attacker can now control execution of the program.

1.4.3 Heap corruption. Like the stack overflow, heap corruption occurs when dynamically allocated memory in the heap is susceptible to a buffer overflow. However, heap memory is allocated differently than stack memory (Dowd et al., 2006). Since heap memory is dynamically allocated there is memory management algorithms that control the management of this memory. Heap memory typically contains additional information about its characteristics, relationship to other blocks of memory and information about its state. Free nodes of memory are typically managed in a singly or doubly linked list and each node contains pointers to the next and previous nodes (Dowd et al., 2006). At certain times, free memory is coalesced into larger chunks to reduce fragmentation in heap memory. If an attacker can overwrite heap memory they have the

`

potential to corrupt this meta-data (the list pointers) and abuse maintenance algorithms to gain control of the program. They can also corrupt data in memory such as values, objects (to include virtual function tables) and function pointers. These conditions can lead to arbitrary code execution, corruption of valid data, denial of service and other vulnerabilities.

1.4.4 Use-After-Free. A use-after-free (UaF) vulnerability occurs when a region of memory that has been allocated is prematurely freed (Dowd et al., 2006). Since freed memory can then be reallocated, there is potential for an attacker to request and fill that memory with attack data (i.e. shellcode, ROP chain, change valid data). Since the memory was prematurely freed, a valid pointer still references that area of memory. Depending on the use of the memory in the program, if the program attempts to use the previously freed block of memory it can lead to arbitrary code execution, corruption of valid data, data leakage and premature program termination, just to name a few.

1.4.5 Double-Free. A double-free occurs when a free operation is called twice on the same memory address ("CWE-415: Double Free," n.d.). This condition can potentially lead to a buffer overflow in which an attacker can overwrite memory with attack data.

1.5 Evolution of Memory Corruption Mitigations

Nominally, memory corruption attacks began with the stack overflow and the release of “Smashing the Stack for Fun and Profit” by One (1996). This brought into view the vulnerable nature of software and the underlying machine architecture. The response to memory corruption

`

was twofold: focus was given on developing more secure software and computer component manufactures began looking for ways to mitigate such vulnerabilities at a hardware level.

The first significant mitigation came in the form of the no-execute bit (Krahmer, 2005). Initially, the stack was an executable region of memory, meaning that if an attacker could abuse a buffer overflow they could use this region of memory to stage arbitrary code and design for its execution. The no-execute bit was a hardware-enforced change that made the stack region of memory non-executable (Krahmer, 2005). This change would prevent attackers from leveraging an overflow to achieve code execution. While effective in stopping the original stack overflow vulnerability, attackers could circumvent this defense by placing their attack code in heap memory and then corrupting the stack to point to this allocation.

Since heap memory is a different region in the process' virtual address space it was not subject to the implementation of the no-execute bit as the stack was. One difficulty in mitigation technology is the rate at which it is adopted. Microsoft did not introduce support for the NX features until Windows XP Service Pack 2 in 2004 and Windows Server 2003 Service Pack 1 in 2005, nearly 8 years after One (1996) published "Smashing the Stack for Fun and Profit" ("Executable Space Protection," n.d.). Microsoft commonly referred to this as data execution prevention (DEP) and through this feature a program could control which memory pages where allowed to contain executable code.

As is prevalent in this field of study, the introduction of mitigation technology began a back and forth in the development of exploit mitigation and attack techniques. The next mitigation introduced is commonly referred to as a stack guard (stack cookie, canary) (Cowan et al., 1998). Overall, the purpose of the stack guard was to place a known (random) value after a stack-based buffer that is checked before a function returns. If the value has been modified than the operating

`

system assumes an overflow and can terminate the program instead of continuing to execute code (i.e. by returning from the function). The cookie value was determined at runtime and therefore not susceptible to a priori knowledge. This feature was added to compilers and therefore did not require any modifications to existing code, code did have to be re-compiled with a supporting compiler though and supported by the operating system. Figure 4 depicts a stack frame susceptible to a stack overflow but with the stack guard implemented, the stack guard is located at *EBP-4* and labeled “cookie”.

While effective, attackers found a way to reliably bypass this defense through a technique called Structured Exception Handling (SEH) overwrite. While this attack begins with a buffer overflow, instead of overwriting the return address in the function’s stack frame, the attacker overwrites a significant portion of the stack and triggers an exception. The exception handler mechanism would engage by retrieving function pointers responsible for handling exceptions from the stack. Since the function pointers are stored in the stack, the attacker can overwrite these function pointers. When the exception handler dispatches the exception, it uses a corrupt value and the attacker can gain control of the program. The response to this attack was called SafeSEH, which checks function pointers before they are used to ensure they have not been corrupted ("Executable Space Protection," n.d.).

`

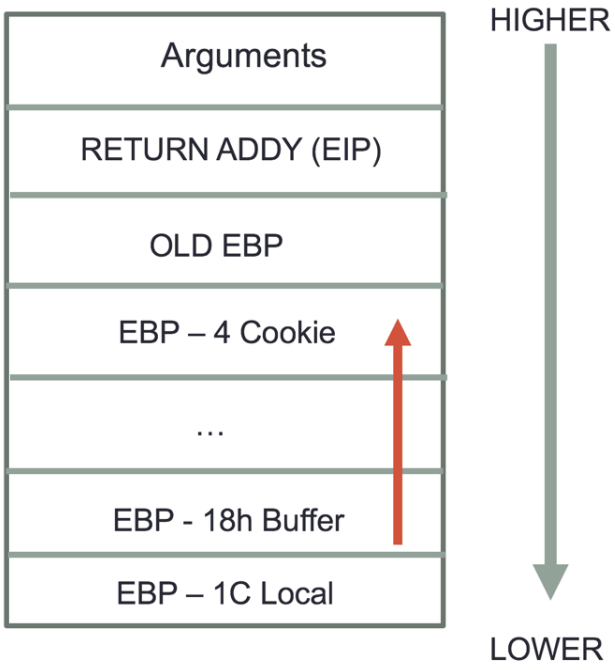


Figure 4. Stack frame with stack guard protection.

After the introduction of the NX bit, and later DEP in Microsoft Windows, attackers had to find other techniques to bypass non-executable memory. The first technique adopted was termed "return to libc" (Krahmer, 2005). This technique was later built upon and is now commonly referred to as return-oriented programming (ROP), which is depicted in Figure 5. There exists considerable overlap in the return-to-libc and ROP techniques and only ROP will be discussed in this section. This technique subverts non-executable memory by not relying on executable code to be placed in those regions of memory. Instead, the attacker pieces together individual sets of instructions from the program and any loaded modules that end in a RET instruction. Since the instructions end in a RET, execution always flows back to the stack to retrieve the return address (a RET instruction assumes that the stack pointer ESP is pointing to the return address and moves that value into EIP, the instruction pointer) (*Intel 64 and IA-32 Architectures Software Developer's*

`

Manual, n.d.). The sequences of RET instructions is referred to as gadgets and an attacker's payload can consist entirely of these gadgets. Since the instructions are borrowed from executable regions of memory, an executable stack is not needed.

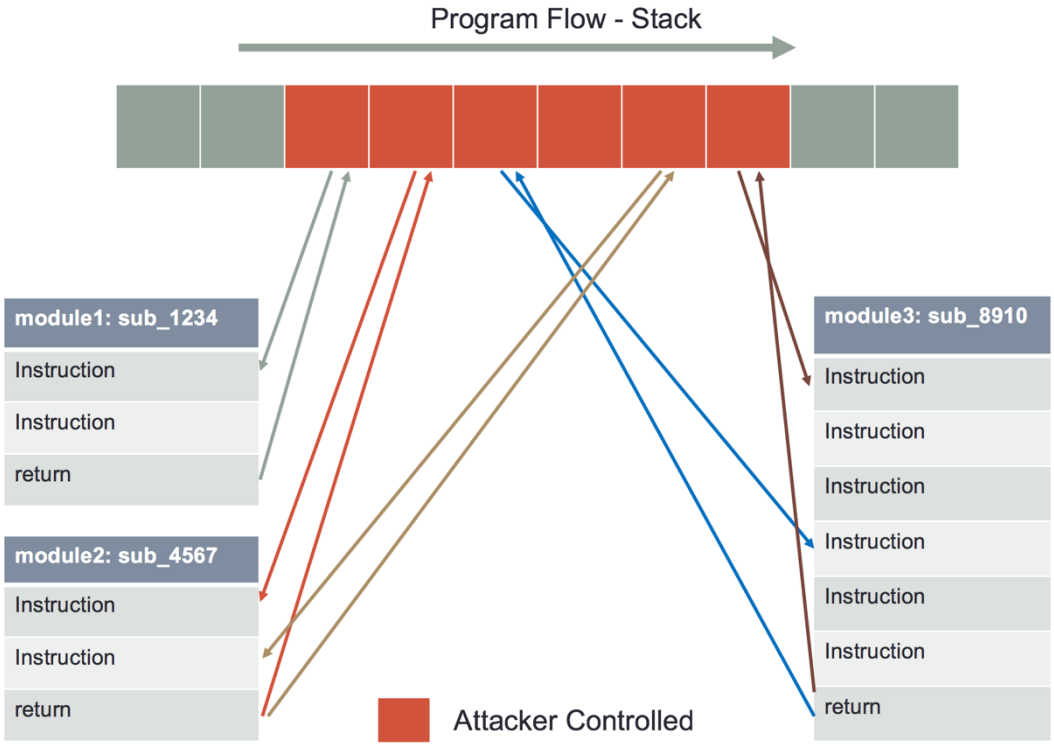


Figure 5. Execution of a ROP chain.

ROP gadgets can be found not only in the code of the program, but also in the libraries loaded into the program's virtual address space. The process of finding a gadget involves searching executable code regions for a RET instruction and moving an arbitrary number of bytes higher in the address space from the address where the RET instruction was found. It is arbitrary since

`

different lengths can result in different instructions. Figure 6 demonstrates how a single RET instruction can be used to create multiple instructions.

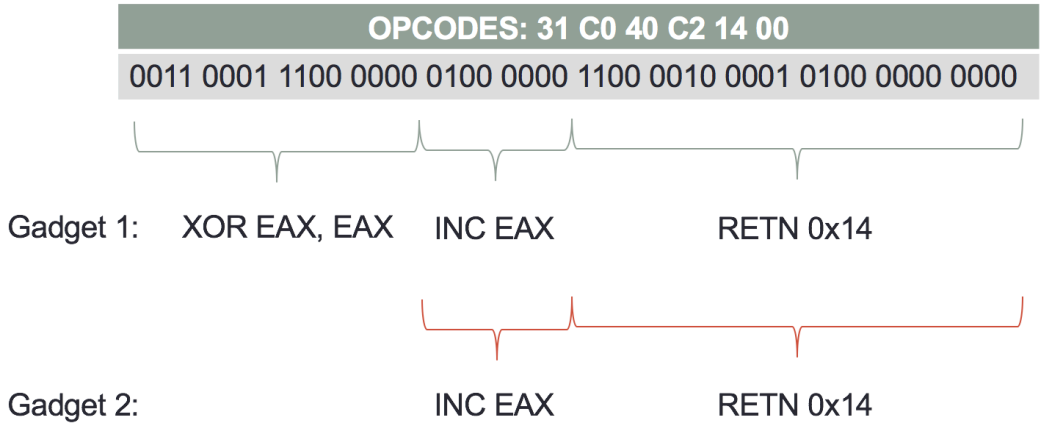


Figure 6. Varying instructions from number of bytes used from RET instruction.

A series of ROP gadgets is referred to as a ROP chain, and this becomes the primary payload for an attacker. With a ROP chain, the attacker is then able to utilize ways in which to disable or circumvent executable space protections. For example, an attacker's ROP chain may re-mark a page as executable or allocate a new page with execute permissions. As part of the original proposal, address space layout randomization (ASLR) was to mitigate this type of attack and later adopted in modern operating systems ("Executable Space Protection," n.d.). Attackers that leverage ROP rely on the predictability of the addresses in which the gadgets they need are located. Before ASLR, programs, and their imported libraries, would be loaded at predictable virtual addresses. ASLR ensures that each time a program is run random addresses are used. This complicates an attack using ROP since the addresses used for gadgets will change with each execution of the program, either breaking the ROP chain or changing its behavior. In combination with DEP this has proven to be an effective layering of defenses. However, attackers have still been able to circumvent ASLR by locating address leaks in a program. If an attacker can reliably

`

find a base address for a loaded library, they still have the potential to construct a ROP chain and bypass DEP and ASLR protections.

Another form of memory corruption attack involves indirect calls. An indirect call is when a call instruction uses a dynamic value as the target. This value is created during runtime and often stored in memory, which gives an attacker the ability to corrupt the target address before it is used. Microsoft introduced a feature termed "Control Flow Guard" (CFG) with Windows 8.1 Update 3 and Windows 10 in November of 2014 (Microsoft, n.d.). This feature checks an indirect call target before it is used by the call instruction, triggering program termination if it determines that the value is not a valid call target.

1.5 Dissertation Organization

This dissertation is organized in the following chapters. This chapter provided the reader with an introduction to the problem and related background on topics pertinent to this work. Chapter 2 provides a literature review to introduce the reader to the current state of binary analysis. Chapter 2 reviews current binary analysis frameworks, complications inherent with binary analysis and proposed techniques for effective binary analysis. Chapter 2 will also cover shortfalls in binary analysis frameworks and how the proposed work addresses those gaps. Chapter 3 discusses the research approach taken for this work and development of the framework. Chapter 3 also includes discussion relevant to the design decisions made while constructing the framework and how this framework differs from previous work. Chapter 4 discusses the results of the framework and highlights the novel search architecture developed. Finally, Chapter 5 provides a summary and proposes avenues for future work.

`

CHAPTER 2

LITERATURE REVIEW

This chapter provides the reader with background on binary analysis and other related topics important in understanding the need for the framework. Section 2.1 covers background information related to generating an executable program. A core activity of the framework will be in producing disassembly, which is the output from interpreting the data within an executable program. This provides a foundation for creating an effective platform for binary analysis and is the topic of section 2.2. Section 2.3 introduces instruction set architectures, which define the binary machine code interpreted by a central processing unit (CPU) as instructions. A framework that performs disassembly requires an understanding of instruction set architectures as the disassembly output is generated by interpretation of the machine code. This section also covers nuances in the prevailing instruction set architectures that can complicate the process of disassembly and lead to inaccurate or misleading output. Section 2.4 discusses recognized techniques for performing the disassembly of machine code and the drawbacks associated with each approach. Before disassembly can be performed on a binary file, the area of the file that contains executable code must be identified. Section 2.5 provides readers with an overview of binary file formats. Finally, section 2.6 reviews binary analysis frameworks. This section will discuss their features, what they lack and how the framework will address those shortfalls.

2.1 Generating an Executable Program

The process of generating an executable program involves the translation of source code, written in a high-level language such as C or C++, into object code. Object code represents the

`

original instructions but in a binary form, which is required for execution by a central processing unit, or CPU. The final step is to perform linking, which resolves external symbol references for the resulting binary (Allain, n.d.). While the definition of a symbol can vary slightly, in this context a symbol represents a function or variable located in an external module ("Linker (computing)," 2016).

2.1.1 File formats. Each operating system defines the format in which this binary data is stored. Microsoft defines the COFF format for object files, and the portable executable (PE) file format for executables (EXE) and dynamically-linked libraries (DLLs) (Microsoft, 1999). While the COFF and PE file formats are similar, an object file will lack the final step of resolving external symbols and therefore not be ready for execution by the operating system. It is the latter form that libraries and executable programs are distributed and often the target of reverse engineering activity. For executable programs, the PE file format allows the operating system loader the ability to determine how to map the binary data into memory, relocate the binary, and begin executing at the appropriate entry point (Russinovich, Solomon, & Ionescu, 2012). Similarly, for dynamically-linked libraries the PE file format allows for the library to be loaded into the address space of an appropriate executable by the operating system during program execution.

2.2 Static Binary Analysis

According to Sikorski and Honig (2012), at the core of the reverse engineering process is the analysis of binary files to determine the functionality of the source program. Reverse engineering stems from two complementary but distinct activities, static and dynamic analysis (Popa, 2012). Dynamic analysis techniques involve the execution of code and the monitoring of program behavior to determine the functionality of the target program, while optionally exploring

`

for vulnerabilities. Static techniques involve the analysis of binary files without execution and require disassembly or decompilation of the target software in order to determine functionality (Eagle, 2011). According to Eagle (2011), a disassembler "undoes the assembly process, so we should expect assembly language as the output (and therefore machine language as input)" while a decompiler will attempt to "produce output in a high-level language when given assembly or even machine language as input." Determining program functionality has proven to be a complex subject matter and is an undecidable problem (Wartell, Zhou, Hamlen, Kantarcioglu, & Thuraisingham, 2011).

2.3 Instruction Set Architectures

Schmalz (n.d.) discusses the instruction set architecture (ISA) and defines it as the machine language that can be interpreted by a computing machine. High-level languages are compiled into the supported binary formats for a particular operating system. The compilation of an executable program results in the loss of meaningful information from source code as it is discarded as unnecessary in the resulting machine code. This includes such information as function names, variable names and the ability to correspond disassembly output to the original high-level code ("Linker (computing)," 2016). This adds complexity when performing reverse engineering activities, as this information must then be determined by the reverse engineer in its absence.

2.3.1 Differentiating code and data in ISAs. Challenges also arise from the organization of the machine code itself. The portable executable file format allows for segregation of data and instructions. However, the Intel ISA allows for the mixing of code (i.e. instructions) and data. Other ISAs, such as RISC and Java byte-code, keep code and data isolated from each other (Wartell et al., 2011). In addition, Intel x86 also uses variable-length and instruction

encodings that are unaligned (Wartell et al., 2011).) Wartell et al. (2011) discussed the complexities with the x86 instruction set and proposes an algorithm for differentiating data from code in x86 binaries to increase the accuracy of the disassembly process. Data inaccurately treated as instructions will lead to incorrect disassembly results and any analysis derived from the inaccurate instructions. To overcome this, the author's algorithm isolates byte sequences and then performs analysis on those sequences using machine learning. This process further allows the bytes to be classified as data or code, effectively identifying data that may potentially be misinterpreted as code. The authors provide results produced from selected samples but the overall effectiveness of the proposed method is not determined.

2.4 Prevailing Methods of Disassembly

Creating accurate representations of binary programs through the process of static disassembly is a problem that has yet to be solved (Andriesse et al., 2016). Two prevailing techniques exist when performing the analysis of code during static disassembly: linear sweep and recursive traversal (Linn & Debray, 2003). Linear sweep simply begins at the first byte of executable code and begins disassembling, in a linear fashion, forward. In comparison, recursive traversal begins at the first byte, but instead of disassembling in a linear fashion, follows the control flow of the program. It does this by analyzing any control flow instructions, such as *call* or *jmp,* and determining the possible locations for those instructions. These locations are disassembled, and any control flow instructions contained within are also analyzed. This process repeats recursively until the binary is fully analyzed. However, these approaches each pose their own unique complications. Linear sweep is unable to account for data intermixed with the instructions, which may lead to the disassembly of data as executable code and therefore incorrect results.

`

Recursive traversal encounters complications when identifying all the appropriate target locations of control flow instructions and may miss sections of executable code.

2.4.1 Disassembly complications. Paleari, Martignoni, Fresi Roglia, and Bruschi (2010) also discussed the complexities inherent in the disassembly process. In their work the authors evaluated eight disassemblers targeting the Intel x86 platform to determine the correctness of each disassembler. They developed a methodology they termed *N-Version Disassembly* in which they compare the output of the disassemblers used to detect anomalies. Their methodology was then utilized to compare the disassemblers and bugs in the disassembly process was identified in each of them. The identified bugs were due in large part to the complexity and ambiguities of the Intel instruction set and how each compiler was implemented to handle them. This work highlights the complications inherit with disassembling the x86 instruction set. The authors approach focused on the accuracy of the disassemblers, calling into question their accuracy.

To further complicate the disassembly process, new techniques continue to emerge to complicate or disrupt the disassembly process, whether by security researchers or malicious actors. Jamthagen, Lantz, and Hell (2013) proposed a new technique for complicating the disassembly process from a binary source. Their approach utilizes the ability to provide arbitrary values within a 9-byte no-operation (NOP) instruction, taking advantage of the variable length property of the x86 ISA discussed by Wartell et al. (2011). This allows for additional paths of execution inside a single program, a main execution path (MEP) and a hidden execution path (HEP). The path executed can depend on the external environment, such as the presence of virtualization software. By utilizing the HEP, disassembly can produce incorrect results and lead the analyst to wrong conclusions or make incorrect assumptions. The authors demonstrate the viability of this approach using sample programs.

`

2.5 Binary File Formats

Closely related to the disassembly process is the PE file format, as it represents the format in which the disassembler should begin parsing executable instructions and not data. Hahn and Register (2014) discusses the PE file format in detail as well as how malware authors often abuse this file format to complicate the disassembly process. The prevalence of malware distributed utilizes the PE file format due to the popularity of the Microsoft Windows series of operating systems. To increase their effectiveness, malware authors introduce malformations that allow the program to still be executed by the OS but complicates static and dynamic analysis tools used for detection and discovery of program functionality. The author's framework successfully analyzed and identified a large sample of malware, 103,275 samples, as well as 269 malformed samples. This work details the PE file format and numerous techniques used by malware authors to violate the PE specification yet still produce executable programs that will be handled by the operating system. In contrast, the research done by Gawlik and Holz (2014) focuses on the implementation of the code instead of the file format that contains the executable code.

2.5.1 File format malformations. Malware authors also utilize encryption or obfuscation to hinder the analysis process. This can become especially important when analyzing system libraries that, unknown to the analyst, may be artifacts from a malicious attack. Zwanger, Gerhards-Padilla, and Meier (2014) discussed methods that can detect x86/x64 code regardless of file type and obfuscation attempts. This technique focuses on the byte-level instead of assembly mnemonics produced after disassembly. As the work by Paleari et al. (2010) demonstrated, the disassembly process is not guaranteed to be absolutely correct, leaving room for errors during analysis. By focusing on the byte level the detection technique proposed is not dependent on the disassembly process. The authors show through a rigorous evaluation process that their technique

`

can reliably identify code in an arbitrary binary file. In addition, this method could be used to identify errors in disassemblers which, in turn, could be used to improve the process of disassembly.

2.6 Binary Analysis Frameworks

Binary analysis frameworks have been created to streamline the process of performing binary analysis. These frameworks often exhibit common characteristics in performance, support and output provided to the end-user. This section will discuss prevailing frameworks, features provided and limitations. The section concludes by identifying how the Binary Analysis Framework addresses these limitations.

2.6.1 PEV. Efforts have been made to stream-line the process of reverse engineering, whether under a static or dynamic context. PEV was introduced by Merces and Weyrich (2017) as a "fast, scriptable, multiplatform, feature-rich, free and open-source" (Merces & Weyrich, 2017) framework for the analysis of portable executable files. In addition to providing an automated framework for parsing and displaying information about a PE file, it also provides functionality to assist in identifying malformations, indicators of potential malicious activity, and the ability to generate disassembly. While a versatile platform, it is limited to parsing PE files, does not persist any data and does not provide an architecture for searching the instructions of the program under analysis.

2.6.2 BARF. A framework that aims to be multi-platform, BARF (STIC, 2017) has also been released as an open-source project. BARF consists of three components: the core of the framework, architecture support and analysis functionality. At the core of the architecture support is the use of the Capstone disassembly engine. This framework provides broader support than that

`

offered by Merces and Weyrich (2017) in PEV as it supports multiple file formats, supports the ARM architecture, utilizes an intermediate language and offers a satisfiability modulo theories (SMT) solver. Tools that utilize the BARF framework have also been developed and include the ability to enumerate return-oriented programming (ROP) gadgets, produce control-flow graphs of selected functions and generate call graphs of a selected function. Limitations are like those highlighted in the previous framework, namely there is no persistence model, lacks arbitrary search of disassembly output and does not create a project-based profile that links related binaries.

2.6.3 Angr. Angr is a python-based binary analysis framework ("angr," n.d.). Angr differs from the previous frameworks in that it provides both “static and dynamic symbolic analysis” ("angr," n.d.). A framework that provides symbolic analysis requires additional levels of translation of disassembly output before allowing for user interaction and providing output (Stephens et al., 2016). As with the previous frameworks, interaction with Angr begins with binary analysis. The output from this stage is the disassembled machine code. Frameworks such as BARF and PEV will cease program interpretation at this point and allow for user interaction with the output, it is up to the user to derive meaning from the disassembly. However, Angr provides additional stages of processing before finishing binary analysis to include the use of an intermediate representation of machine code to provide multi-architecture support. Angr additionally provides a solver engine, machine state emulation, program path analysis, semantic representation and full program analysis. The Binary Analysis Framework does not intend to provide a feature set like Angr, but to expand analysis capabilities on disassembly output like frameworks such as PEV and BARF.

2.6.4 Limitations addressed by the Binary Analysis Framework. The Binary Analysis Framework attempts to address several of the limitations discussed earlier in this section.

`

The Binary Analysis Framework is not intended to provide features or address limitations in frameworks such as Angr, frameworks that employ symbolic or dynamic symbolic analysis. Instead, the Binary Analysis Framework was designed to address limitations in frameworks like PEV and BARF, frameworks that produce as the final output disassembly. The primary interaction with such frameworks is in analyzing this output, which provides the context in which the analyst derives program meaning and behavior.

The Binary Analysis Framework will address several key limitations in current binary analysis frameworks. A data persistence architecture will be developed using an open-source database system. The proposed benefits of this design are: reduced time in performing program analysis, an architecture for grouping related binaries and a novel search architecture based on disassembly output. The development of a relational database for data persistence allows for the program to load data from programs already disassembled. This contrasts with having to perform the disassembly of the input file each time a program analysis is desired.

Related input files can easily by grouped together in a project construct. This increases the efficiency of analysis as all related input files can be accessed from a singular graphical user interface. Current frameworks require an instance of the analysis program to be open per file, increasing time requirements in switching between these instances.

The last feature is that of a search architecture that allows for searching by assembly instruction (or disassembly output). This allows for the construction of novel search algorithms that utilizes string comparison functions, regular expressions and database language features. In addition, searches can be performed simultaneously across multiple input files. This allows for an analyst to perform comparative analysis with disassembly output instead of relying on byte-sequence matching. This will expand the scope in which analysts are able to analyze a program

`

by providing a broader facility for search. This builds upon current limitations, which are limited to performing a binary conversion of the input search patterns before performing the search. The result of this architecture will be that of a more comprehensive evaluation of the target input file, allowing for discovery of unknown vulnerabilities and, ultimately, more secure software.

2.7 Literature Review Summary

This chapter has provided background information on generating an executable program, binary analysis, instruction set architectures, prevailing disassembly techniques, binary file formats and binary analysis frameworks. These concepts form the foundation for the process of reversing engineering software and are used as foundational reasons for the proposal of the framework. Research methodology and artifact design are discussed in the next chapter.

`

CHAPTER 3

SYSTEM DESIGN

This chapter offers a formal definition of the Binary Analysis Framework, discusses each component of the framework and details how it was constructed. Novel search functionality is enabled by this design as comparative searches can be performed using database query languages, string comparison functions and regular expressions using assembly language syntax. This approach differs from prevailing binary analysis frameworks in two ways, the use of a relational database for data persistence and the ability to use assembly language syntax to perform searches. Existing frameworks, covered in Chapter 2, require a translation of assembly language syntax into binary instructions before searches can be performed. This creates new avenues to perform analysis of a disassembled program, increasing the understanding of the program logic by the analyst and reducing the amount of time needed to ascertain that understanding. This, in turn, can lead to the discovery of vulnerabilities in the software.

3.1 Research Approach

This work followed the principals of design science as introduced by Hevner, March, Park, and Ram (2004). According to Hevner et al. (2004), design science research is a paradigm that "is fundamentally a problem-solving paradigm. It seeks to create innovations that define the ideas, practices, technical capabilities, and products through which the analysis, design, implementation, management, and use of information systems can be effectively and efficiently accomplished". Additionally, Hevner et al. (2004) defines design science as "a body of knowledge about the design of artificial (man-made) objects and phenomena— artifacts designed to meet certain desired

`

goals." The focus of the research is on the study of phenomenon, not necessarily naturally occurring, in which the research contributes to the corpus of knowledge related to that phenomenon, or certain behaviors or characteristics of that phenomenon. Therefore, research activity is focused around the development of an artifact, in which there is a higher tolerance going into the research for failure to provide for the exploration of the unknown.

3.2 Limitations

The Binary Analysis Framework will take an optimistic approach to validating the input files. Malware authors routinely abuse the PE file format to bypass detection and perform other nefarious activity on a computing system (Hahn & Register, 2014). The framework will not attempt to detect intentional malformations and may be unable to parse the desired file or produce incorrect disassembly listings. Consideration has been given to this problem and the framework will support future expansion of more robust file validation and will be discussed later in this section.

Code obfuscation is a technique employed in executable code to make the inner workings of the software difficult to understand (Linn & Debray, 2003). Code obfuscation comes in many forms and can include packing, dynamic code generation and the use of shellcode as well as encryption routines (Sikorski & Honig, 2012). Code obfuscation is used to achieve a variety of goals, such the protection of intellectual property within the software or to inhibit the discovery of software vulnerabilities that could be leveraged by an attacker to exploit the software.

Code obfuscation is also commonly used by malware authors to hide the intended functionality of their program and to evade signature-based detection by anti-malware products (You & Yim, 2010). Regardless of the intent, these techniques make the process of disassembling

`

obfuscated software complex, as the data that the disassembly process encounters may not be executable code but encrypted or obfuscated data. To disassemble this code correctly it would need to be de-obfuscated and with the number of possible ways in which the code can be obfuscated, quickly becomes a daunting task (Linn & Debray, 2003). The Binary Analysis Framework will not be able to support the disassembly of obfuscated or encrypted code due to these difficulties.

3.3 Code Organization

The Binary Analysis Framework is a Java application which utilizes Java SE and JavaFX. Java SE version 1.8 and JavaFX version 8.0.111-b14 were selected for the development of this framework, as they were current version of Java SE and JavaFX at the time of development. There are four platforms available from Oracle, the developer of Java, that can be selected when building a Java-based application: Java SE, Java EE, Java ME and JavaFX. Java SE, or Standard Edition, is the most prevalently used platform when developing desktop applications. Per Oracle, "Java SE's API provides the core functionality of the Java programming language. It defines everything from the basic types and objects of the Java programming language to high-level classes that are used for networking, security, database access, graphical user interface (GUI) development, and XML parsing" (Oracle, n.d.). Other platforms include Java EE, which is designed for enterprise development of "large-scale" applications and Java ME, which is intended to support mobile devices (Oracle, n.d.). JavaFX was also utilized and provides core libraries for developing graphical user interfaces.

A key consideration for the selection and use of Java was for the portability of the application. Compiled Java differs from languages such as C and C++ in that Java is compiled

`

into an intermediary byte-code, which is then executed by a Java Virtual Machine (JVM) installed in the host operating system ("Java Virtual Machine," n.d.). The JVM is responsible for providing translation from byte-code to machine code for the specific architecture of the host. In comparison, languages such as C or C++ are compiled directly into machine code for the intended platform. If multiple platforms are to be supported than a corresponding number of compiled executables must be provided by the developers or source code made available from which an executable file can be compiled individually. The flexibility of the Java architect allows for Java applications to be distributed as a single application file, if there is a supported JVM the Java application will be able to execute on that host.

Java applications are logically organized by project constructs. A Java project consists of custom code, third party libraries and libraries provided by Java SE and other resources needed by the application under development. Java project code is organized in two ways: through physical separation of the project files using folders and the logical organization of code through packages. Java packages allow for the organization and reference of code through naming conventions, which provides code organization as well as the ability to access specific code by name ("Namespace," n.d.). The use of packages also allows for a modular design of the framework, which intends to increase organizational effectiveness of maintaining the code as well as to support the modular growth of the framework for future expansion.

3.4 Overview of the Binary Analysis Framework Components

The overall design of the proposed framework is depicted in Figure 7. The Binary Analysis Framework will have four primary components: input file validation, file parsing and disassembly engine, data storage architecture and the graphical user interface. The process of analysis will

`

begin with the loading and parsing of the desired software. Initially, the Binary Analysis Framework will only support 32-bit binaries in the PE file format. The support of 32-bit PE files is not a design limitation but a decision made by the authors to limit the scope of the work, the modular design of the framework will allow for future expansion to include 64-bit binaries as well as additional instruction set architectures (ISA), such as ARM.

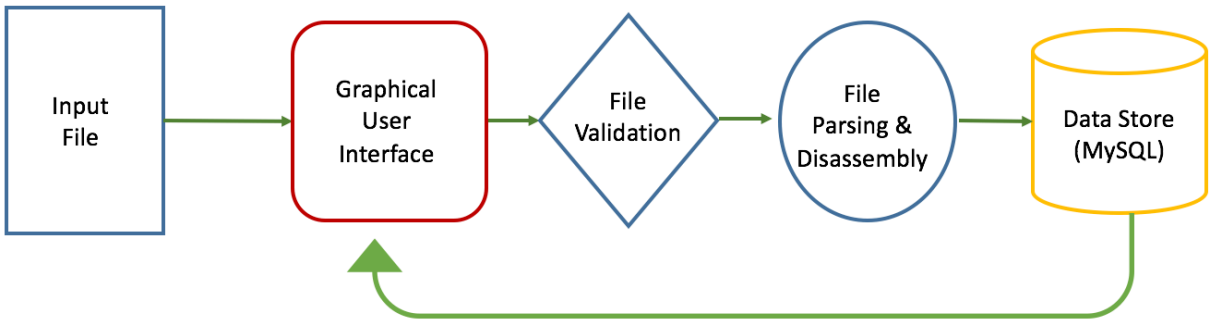


Figure 7. Overview of the Binary Analysis Framework components.

Interactions with the framework will begin by selecting an input file, which will be analyzed to determine file format. If the format is supported by the framework then it will be furthered analyzed to confirm that the file conforms to its defined specification, such as the PE file format specification provide by Microsoft (1999). Due to the focus on the PE file format for this work, the input file will be analyzed to determine a valid 32-bit PE file. If an invalid PE file is detected it will not be sent to the next stage of the framework for disassembly and analysis will cease.

Upon determination of a valid PE file, the framework will direct the input file to the next stage, the file parsing and disassembly engine. The results of this stage will include the characteristics of the file through the parsing of the files header information as well as identification of the code sections. Identified code sections will be disassembled and the output

`

stored in a relational database provided by the database architecture. The database will then be utilized to provide the data for display in the graphical user interface, which allows for analysis of the imported file. An example of characteristics would be the use of data execution prevention (DEP), address space layout randomization (ASLR), the use of stack cookies and control-flow guard (CFG). After successful parsing and disassembly, the data will be directed to the data storage mechanism for permanent storage.

The Binary Analysis Framework will process the results of the disassembly engine to match the schema of the data store. This information will be stored in a relational database using MySQL 5.7.17. The schema for the data model will be designed to allow for granular inspection of the individual instructions and the characteristics of each imported file. This provides not only a storage mechanism for the currently imported file, but also a long-term storage mechanism for each file that is imported into the Binary Analysis Framework. The desired results of this architectural design are twofold: to allow for novel search techniques to be developed by utilizing a relation database and to avoid the need to disassemble the input file each time binary analysis is to be performed.

Additionally, the Binary Analysis Framework will allow for the creation of projects, in which software that is related can be grouped together in a logical collection, as defined by the user. This allows for increased efficiency in exploring the imported software as well as instruction-level comparison between all imported files within a project. The database model will be utilized to develop the frameworks search functionality and utilized the disassembly output rather than performing binary searches on the software's machine code. This allows for a higher level of abstraction when analyzing software, searching by assembly instruction and operands instead of binary or byte sequences.

`

3.4.1 Graphical user interface. Initial interaction with the Binary Analysis Framework will begin with the graphical user interface (GUI). Upon application start, the user is presented with the primary user interface presented in Figure 8.

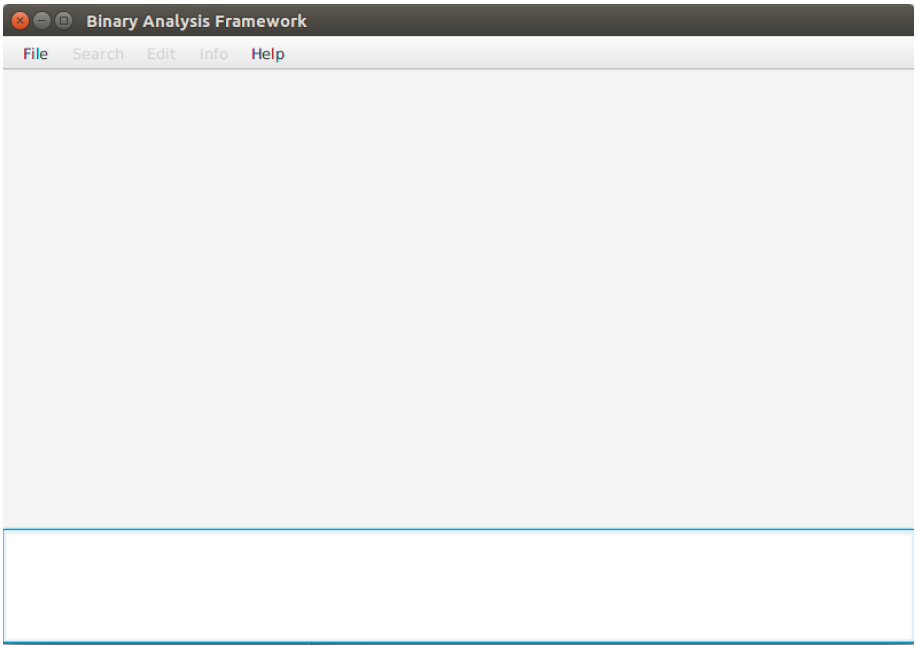


Figure 8. Primary user interface of the Binary Analysis Framework.

The graphical interfaces are designed using an FXML document, as defined within the JavaFX standard ("4. Using FXML to Create a User Interface," n.d.). FXML is based upon extensible markup language (XML) and provides for the development of the user interface components. It is desirable to provide a decoupling, or abstraction, in the development of software applications to increase the ease at which the software can be maintained and further developed (Mo, Cai, Kazman, Xiao, & Feng, 2016). This framework endeavors to follow in such a pattern. To achieve this, the functionality associated with the elements of the user interface are decoupled and organized in a series of *controller* classes, each interface corresponds to a unique controller class. This design pattern allows for the clear distinction in the application code between

`

individual interfaces and their corresponding application logic and is used throughout the development of the framework.

To begin using the framework, the user must create or load a project. Project management is accomplished by expanding the *File* menu option and selecting the *Load Project* menu item. The user is then presented with a dialog listing all previously created projects, in order of frequency of use, as well as options to create a new project. The dialog presented is shown in Figure 9.

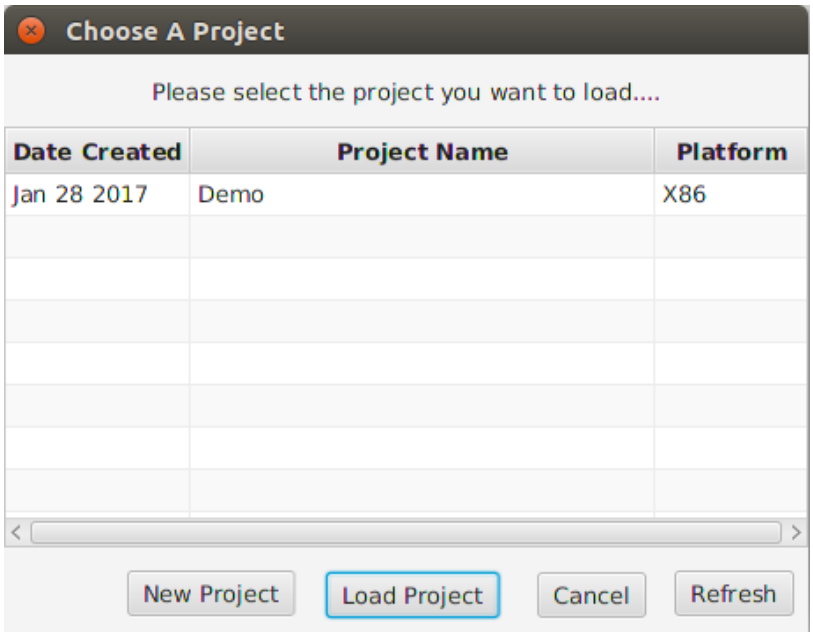


Figure 9. Project management dialog.

The concept of the project is to provide a logical grouping between imported software for analysis. For example, a project could be created to analyze a closed source application. Included in this project would be the target application as well as any software dependencies. Another example would be that of an operating system profile, in which software related to specific versions of an operating system could be created. Creating a new project requires that the user provide a name for the project as well as to select the target platform. The target platform is a statically

generated list of values retrieved from the database and is used for display purposes only. The selection of a target platform does not have an impact on the analysis, parsing or disassembly of any input files. Future work would involve expanding this listing to provide a dynamically generated listing of all supported platforms by the Binary Analysis Framework.

Once a project has been loaded, a series of file menu options along the top portion of the primary interface become available. If the project contains imported software, tabbed content displaying the disassembly output and identified functions of each import will be generated. At this point the user can resume analysis on the previously imported software or import a new file. To import a new file into the project, the user can select the *Import Binary* menu item from the *File* menu. Upon selection, a file selection dialog will be presented to the user and allow for the selection of the desired file from their local file system. The file selection dialog does not filter the files and folders displayed and is therefore up to the user to select an appropriate file for import.

3.4.2 Input file validation. After selection of the desired input file, the framework will attempt to determine the appropriate file format. This is the stage responsible for identifying the appropriate file format of the provided binary file. Identification of a file format allows for accurate determination of executable code in the file, which ultimately is the information populated in the graphical display and that which the analyst uses to seek out software vulnerabilities and other security issues. Inaccurate data can be displayed to the analyst if the framework is unable to parse the given file format or makes an incorrect assessment of the file type. Limitations in file parsing were discussed in section 3.2. The remainder of this section discusses the code developed to handle this activity.

The controller that supports the primary user interface is called *Controller.java* inside of the *GUI* package and includes functionality to begin the importation, parsing and disassembly of

`

the desired input file. File parsing and disassembly begins by the framework utilizing a custom *DisassembleService* class, which provides several interfaces in which to make the framework modular, allowing for the expansion of additional file formats and architectures. The authors endeavored to use the Abstract Factory Pattern, in which base classes are responsible for creating related objects without knowing the details about the sub-class implementation (Martin, 2009).

As depicted in Figure 10, the *DisassembleService* class exposes a single public method *getCode* on line 13, which returns a *Code* object on line 32. The *Code* object is a general-purpose class whose properties allow for the storage of the disassembled code, function maps and other necessary attributes to facilitate further analysis. The *Code* object is intended to be implemented by sub-classes, following the Abstract Factory Pattern. This allows for a modular design in which the properties and methods of each supported file format remain consistent within the framework. Inside of the *getCode* function an instance of a *LoaderFactory* class is instantiated at line 15.

```
13    public static Code getCode(File file) {
14
15        LoaderFactory factory = new LoaderFactory();
16        LoaderInterface loader = factory.getLoader(file.getAbsolutePath());
17        DisassemblerFactory disFactory = new DisassemblerFactory();
18        DisassembleInterface d = disFactory.getDisassembler(loader);
19        Code c = d.disassemble(loader);
20
21        c.setCodeSectionBytes(loader.getCode());
22
23        c.setASLREnabled(loader.isASLREnabled());
24        c.setDEPEnabled(loader.isDEPEnabled());
25        c.setCFGEnabled(loader.isCFGEnabled());
26        c.setStackGaurdEnabled(loader.isStackGuardEnabled());
27
28        c.setCodeSegmentName(loader.getCodeSegmentName());
29
30        d.generateFunctions(c);
31
32        return c;
33    }
34 }
```

Figure 10. Function getCode from DisassembleService.java.

`

From the *LoaderFactory* object instance, the *get Loader* function is called, providing as an argument the path to the input file. It is in this method that the input file is parsed to determine the appropriate file format. As discussed in the limitation section, detection of any file format malformations is not performed. However, it is at this point in the framework that additional parsing and validation could be added to detect for known malformations. To determine the file format, the framework checks for known file format signatures, also known as magic numbers ("File Format," n.d.). In the PE file format, the first two bytes of the file are the ASCII characters 'M' and 'Z', or 0x4d and 0x5a in hex, and represent the magic number. Figure 11 provides the implementation of the *getLoader* function.

```
10        public LoaderInterface getLoader(String path)
11        {
12            LoaderInterface result = null;
13            byte[] originalBytes = LoaderInterface.loadBytes(path);
14            if (originalBytes[0] == 'M' && originalBytes[1] == 'Z')
15                {
16                    result = new PE();
17                }
18            else if (originalBytes[0] == 0x7F) // elf
19                {
20                    result = new ELF();
21                }
22            result.setBytes(originalBytes);
23            return result;
24        }
25    }
```

Figure 11. Function getLoader from x86.java.

This function retrieves the necessary bytes from the input file at line 13 and compares them to known signatures beginning at line 14. Additionally, other file formats can be determined based on their magic numbers and is supported by the framework using a multi-conditional *IF-ELSE-IF* control structure. At line 18 a comparison is made for the hex value 0x7F, which represents the executable and linkable format (ELF) supported in Linux operating systems ("Executable and

Linkable Format," n.d.). However, the framework is unable to parse input files in the ELF file format in its current iteration and was provided to illustrate future expansion of the framework.

Once the appropriate file format is identified, an instance of a supporting class is instantiated. A base class named *LoaderInterface* has been developed to provide for the base properties and methods that will be required of disassembled programs, regardless of file format. Classes that extend this base class will also be created for each supporting file format and implement any abstract functionality to provide the consistency and extensibility of the framework. In the case of a PE file, a new instance of the PE class will be assigned to the *LoaderInterface* object and returned. This technique can be observed at line 16 in Figure 11, with the object being returned by the function at line 23. It is through this design that additional file formats can be defined and returned as instances of the *LoaderInterface* class, with each sub-class providing the appropriate implementation of properties and methods per the corresponding file format characteristics.

The *LoaderInterface* class defines several abstract methods, public methods and public members. One method of importance is an abstract method *getType*. Since the *PE* class extends the base class *LoaderInterface* it must provide an implementation of this method, as would any class that extends *LoaderInterface.* The purpose of the *getType* function is to provide introspection of the input file to determine the targeted architecture, such as Intel's X86 or Holding's Advanced RISC Machine (ARM). Since each file format will include this information in varying ways, each class specific to a file format will need to implement the functionality necessary to determine the architecture type. Figure 12 shows the implementation of the *getType* function for the PE file class.

`

```
124⊖        @Override
125         public CODE_TYPE getType()
126             {
127                 ImageFileHeader ifx = fileHeader.getFileHeader();
128
129                 if (ifx.getMachine() == 0x01c4)
130                     {
131                         return CODE_TYPE.ARM;
132                     }
133                 return CODE_TYPE.x86;
134             }
```

Figure 12. Function getType from PE.java.

The *LoaderInterface* class also defines an abstract method *setBytes.* This method is responsible for storing the original bytes associated with the input file in a public member called *bytes.* For example, in the implementation of this method in the *PE* class, the base class version of this method is called through a call to *super.setBytes*, then the member function *load* is called. Figure 13 depicts the *load* function as defined in the *PE* class.

```
39      try
40      {
41          fileHeader = new ImageNTHeaders(bytes, elf_anew);
42
43          this.characteristics =
44                  fileHeader.getImageOptionalHeader().getDllCharacteristics();
45
46          ImageSectionHeader codeSegement = fileHeader.getCodeSegement();
47          codeOffset = codeSegement.getVirtualAddress();
48          codeSegmentName = codeSegement.name;
49
50          code = new byte[codeSegement.sizeOfRawData];
51          for (int start = 0, i = codeSegement.pointerToRawData; start <
52                  codeSegement.sizeOfRawData; i++, start++)
53          {
54              code[start] = bytes[i];
55          }
```

Figure 13. Function load from LoaderInterface.java.

The function *Load* is responsible for parsing the header information of the related file format to locate the corresponding binary data for the executable code of the input file, represented as byte values, for later disassembly. This can be seen in the *try-block* starting at line 39. It is in

`

this function that the raw binary data is read from the input file's code section and stored in a byte array member *code*. The *for-loop* starting at line 51 is responsible for iterating over the raw bytes and storing them in the *code* member.

3.4.3 File parsing and disassembling. Upon successful input file validation, the next stage is to begin file parsing and disassembly. This stage identifies the binary data in the file that represents the executable code and becomes the foundation for the data displayed to the analyst. Once executable code is identified, it can be disassembled. The output of the disassembly process is the assembly instructions used to populate nearly all displays in the framework. The rest of this section discusses the code developed to complete this activity.

An instance of the *DisassemblerFactory* is created at line 17 in Figure 10. The *DisassemblerFactory* class exposes a single function named *getDisassembler*, which expects an instance of the *LoaderInterface* class. At line 18 this method is called and the previously instantiated *LoaderInterface* object provided as the argument value. The purpose of the *getDisassembler* function is to determine the input file's architecture to provide appropriately created objects for disassembly. Function *getDisassembler* accomplishes this by utilizing the *getType* function from the *LoaderInterface* object, as previously discussed.

Figure 14 depicts the function *getDisassembler,* which creates a new instance of a class appropriate to the architecture of the input file. It does this by creating a multi-conditional *IF-ELSE-IF* control structure in which the conditional uses the return value of the *LoaderInterface's getType* function and a constant value representing the architecture. This code technique can be seen starting at line 17. In the case of a 32-bit PE file, this would be an instance of the *x86* class, which is instantiated at line 25. Additionally, architecture and mode information is defined and

`

later used to create an instance of the *Capstone* class, which will use this information to disassemble the code from the input file. The instance of the *Capstone* class is assigned to a public member of the architecture class, *x86*, at line 26. The function *getDisassembler* returns an instance of the *DisassembleInterface* class, which is a base class that is extended by the architecture specific classes, such as *x86,* at line 33.

```
17      if (l.getType() == CODE_TYPE.ARM)
18          {
19              disass = new ARM();
20              arch = Capstone.CS_ARCH_ARM;
21              mode = Capstone.CS_MODE_ARM;
22          }
23      else if (l.getType() == CODE_TYPE.x86)
24          {
25              disass = new x86();
26              arch = Capstone.CS_ARCH_X86;
27              mode = Capstone.CS_MODE_32;
28          }
29
30      Capstone cs = new Capstone(arch, mode);
31      cs.setDetail(Capstone.CS_OPT_ON);
32      disass.setCapstone(cs);
33      return disass;
```

Figure 14. Function getDisassembler from DisassembleFactory.java.

The *DisassembleInterface* class provides two abstract functions: *disassemble* and *generateFunctions*. The *DisassembleInterface* class also exposes a member function *setCapstone* and a public member *cs*, which is the instance of the Capstone class assigned to in the *getDisassembler* function. As was seen with the implementation and relationship of the *LoaderInterface* base class and the *PE* sub-class, a similar relationship exists between the *DisassembleInterface* class and that of the *x86* class, or any class that extends the *DisassembleInterface* class. This design pattern allows for the extension of the framework to support additional architectures, such as ARM, by allowing for a modular design in the *getCode* function of the *DisassembleService* class. To extend the framework, additional classes that support

the desired architecture must be created and can be done without having to modify the core functionality of the framework.

After an instance of the *DiassembleInterface* class has been created and a value assigned from the function *getDisassembler*, the code from the input file can be disassembled. This is accomplished through the *disassemble* member function of the *DisassembleInterface* class, which was instantiated in the *getCode* function at line 18 in Figure 10. The *disassemble* function expects a single argument, an instance of the *LoaderInterface* class. This value is provided by the earlier instantiation of this object and contains the required functionality to produce the data that represents the code section of the input file. The *disassemble* function is defined as an abstract function in the *DisassembleInterface* class, which is extended and then implemented in the architecture appropriate sub-class. In the case of a PE file Intel's x86 architecture, this would create an instance of the *x86* class. Figure 15 depicts the implementation of the *disassemble* function from the *x86* class.

```
114 public Code disassemble(LoaderInterface l)
115 {
116     TreeSet<x86VisitorIns> instructs = new TreeSet<x86VisitorIns>();
117     Code c = new Code();
118     data = l.getCode();
119     done.clear();
120     TreeMap<Integer, Ins> resultMap = c.getCodeMap();
121     start = l.getBaseAddress();
122     TreeSet<Integer> list = new TreeSet<Integer>();
123     list.add(l.getStartAddress());
124     Ins i;
125     Integer current;
126     x86AddressVisitor visitor = new x86AddressVisitor(tempList);
127     while (!list.isEmpty())
128     {
129         current = list.first();
130         list.remove(current);
131         tempList.clear();
132         i = getIns(current);
133         if (i != null)
134         {
```

Figure 15. Function disassemble from x86.java.

The purpose of the *disassemble* function is to disassemble the binary code identified in the input file, which will then be stored in the database. The *disassemble* function begins by retrieving the code from the input file at line 118 in Figure 15. This is accomplished by calling the *getCode* method from the provided instance of the *LoaderInterface* class. The code requested was populated in the member variable *code* in the resulting *LoaderInterface* class. Once the input file code is retrieved, the *disassemble* function has all the bytes necessary to begin disassembly.

During disassembly, this function also keeps track of the linear, virtual address of the instruction. This information is stored in the database and later displayed by the corresponding assembly instruction. The use of a virtual address allows for a correlation between the disassembled instruction and the location the instruction would appear in memory during program execution. The initial address is determined during parsing of the input file and found in the header information of the corresponding file format. For example, in a PE file this address would be determined by using the *ImageBase* member in the *IMAGE_OPTIONAL_HEADER* structure as defined by Microsoft in the PE File Format specifications ("IMAGE_OPTIONAL_HEADER structure," n.d.). Upon determination of the starting address and retrieval of the input files executable code, the function *disassemble* enters its primary loop at line 127. At line 132 a call to *getIns* is made, which requires a single integer parameter that represents the virtual address for beginning of disassembly.

The function *getIns,* depicted in Figure 16, converts this value from a virtual address to a relative offset from the beginning of the byte array, and does this due to the fact that virtual addresses are only of use when used relative to the virtual memory of a program during execution

`

("Virtual Address Space," n.d.). The calculated offset provides an index to the corresponding program code, which is contained in the previously discussed member *bytes.* At line 39 a *FOR* loop is implemented and up to 20 bytes is then copied into a local byte array, which will be used for the disassembly of the instruction. Once the local byte array is populated, it is then passed to the *disasm* member function from the Capstone object that was created during the call to *getDisassembler,* this is done at line 47. The Capstone function *disasm* will disassemble the bytes provided by the local byte array and return a *CsInsn* object from the Capstone library, which will represent the original machine code as an assembly instruction and provide access to disassembly information such as the instruction mnemonic, the operands and the size of the instruction in bytes.

```
30  private Ins getIns(int current)
31  {
32      Ins i = null;
33      Integer startAddressOffset = (current - start);
34      if (startAddressOffset < data.length)
35      {
36          if (!done.contains(startAddressOffset) && startAddressOffset >= 0)
37          {
38              done.add(startAddressOffset);
39              for (int dataAddress = startAddressOffset, codeAddress = 0;
40                      dataAddress < startAddressOffset + 20;
41                      dataAddress++, codeAddress++)
42              {
43                  if( dataAddress < data.length) {
44                      code[codeAddress] = data[dataAddress];
45                  }
46              }
47              CsInsn instruct = cs.disasm(code, current, 1)[0];
48              i = new Ins(instruct, current);
```

Figure 16. Function getIns from x86.java.

The *CsInsn* object is used to instantiate an *Ins* object, which is a custom object created to provide an interface for the disassembled instruction used throughout the rest of the framework. This approach was selected as to provide extensibility in the instruction object and not be limited

`

to that of the *CsInsn* object provided by the Capstone library. Upon function return, this object is tested to ensure it has been created and, if so, the type of the instruction is determined through a call to *getAddressItem*. The *getAddressItem* function determines if the instruction is a branching instruction, in which case the list of addresses original populated with the start address is added to for the disassembly of the program to be able to continue.

Before completion of the *getCode* function, the member function *generateFunctions* of the *DisassembleInterface* object is called at line 30 in Figure 10. This method requires a single argument, the instantiated *Code* object returned from the call to *disassemble.* This function will explore the disassembled instructions seeking the *call* mnemonic and, if found, add it to another collection which contains the address of where the *call* instruction was found. This functionality allows for all functions identified in the input file to be analyzed.

Upon completion of the *disassemble* function, it returns an object of type *Code.* This object has members that contain collections in the form of TreeMaps of the disassembled instructions as well as all identified functions. This function returns to the method responsible for importing a new binary initially called from *Controller.java*. Once the input file is disassembled, the data can be stored in the relational database.

3.4.4 Data persistence. After successful disassembly, the output is stored in a relational database. MySQL was chosen as the relational database engine, the schema developed can be viewed in Appendix A. MySQL provides not only an open-source database solution, but also a non-proprietary format that accommodates the sharing of project data. Many popular disassembly tools, such as IDA Pro, utilize a proprietary format for storage of disassembled programs. This complicates the process of binary analysis amongst teams as there is no inherent way to synchronize analysis. Proprietary formats also restrict the customization of such frameworks and

`

often limit those desiring custom functionality to limited APIs. The use of an open-source database will allow for development of third-party software, accommodating expansion of the framework beyond just analysis but also collaboration and communication capabilities. The rest of this section discusses the software developed to implement the data storage capabilities in the Binary Analysis Framework.

The process of saving the data begins with the creation of a *project* object, which corresponds to the project table in the database schema. The *project* table provides a one-to-many mapping to the *imports* table using an intermediate table *project_imports.* The *project_imports* table allows for multiple imports per project, but also that a single import could be shared across multiple projects. The *import* table is used to store information about an imported file. From the *import* table relationships exist with several other tables to store the disassembled instructions, characteristics of the import, section information and the bytes of the original imported file. The disassembled instructions are stored in the *instructions* table, which has a one-to-many relationship with the *imports* table. The *characteristics* table contains security information about the imported file: if it was compiled to use ASLR, DEP, CFG or stack guard. This table has a one-to-one relationship with the *import* table. A table named *section* stores the original bytes of the identified code sections of the input file and the name of the section. The *section* table has a one-to-many relationship with the *import* table. The final table is called *original_binary* and contains the entire binary content of the imported file.

Figure 17 depicts the process of object creation and data persistence for an imported file.

`

```
112     Project p = gui.getLoadedProject();
113
114     //Create new import
115     Import binary = new Import();
116     binary.setFileSize(file.length());
117     binary.setName(file.getName());
118     binary.setMD5Hash("hash");
119     binary.setProjectID(p.getID());
120
121     logOutput("New binary imported", MESSAGE_TYPE.INFO);
122
123     this.filename = FilenameUtils.removeExtension(file.getName());
124
125     //Store the original bytes
126     binary.SaveRawBytes(file);
127
128     logOutput("Original bytes stored", MESSAGE_TYPE.INFO);
129
130     Code c = DisassembleService.getCode(file);
131
132     Characteristics chars = new Characteristics();
133
134     chars.setASLREnabled(c.getASLREnabled());
135     chars.setDEPEnabled(c.getDEPEnabled());
136     chars.setCFGEnabled(c.getCFGEnabled());
137     chars.setStackGaurdEnabled(c.getStackGuardEnabled());
138
139     binary.setCharacteristics(chars);
140
141     binary.Save();
142
143     section codeSection = new section();
144
145     codeSection.content = c.getCodeSectionBytesAsString();
146     codeSection.name = c.getCodeSegmentName();
147     codeSection.import_id = binary.getID();
148     codeSection.save();
```

Figure 17. Creating Import and Section objects

Once the selected file for import has been disassembled, the currently loaded project is referenced to obtain the *project_id* at line 119. The project id is an internal value used in the database as the primary key to provide for data normalization. Next, an instance of an *import* class is created at line 115, this object corresponds to the *import* table in the database. Properties of the *import* object are updated and include: the size of the input file in bytes at 116, the name of the file at line 117, the project id at line 119, a list of *ShortenedIns* objects and a list of *ShortenedFunction*

`

objects (not depicted in Figure 17). Referencing the *Code* object returned from the *getCode* function, this object provides a method called *getCodeSectionBytesAsString*, which returns all the bytes of the code section of the input file as a string. An instance of the *section* object is created at line 143, which provides an object-mapped interface with the database. It contains several properties: the name of the section, the id of the import that it belongs to and the content of the section. The table that this object corresponds to provides a one-to-many mapping with the import and allows for multiple sections to be mapped.

Following creation of the import object, the framework then saves the instructions and the functions that were disassembled. During the process of disassembly two *TreeMap* collections are created, one that stores function information and another that stores instruction information and can be seen at lines 150 and 152 in Figure 18. These lists are iterated, creating objects *ShortenedFunction* and *ShortenedIns,* which are then used to represent functions and instructions throughout the rest of the framework. Once the new objects are created they are added to a list collection and persisted to the database. These lists are also used to later update the graphical user interface.

The properties of the *ShortenedIns* object are defined as follows: a string to represent the opcodes, an integer value for the virtual address of the instruction, a string for the instruction mnemonic, a string that represents the operands, a string that represents the virtual address and an integer that represents the opcode size in bytes. These properties where chosen to facilitate information display and for search capabilities on the disassembled instructions. Member functions include getters and setters for these properties.

`

```
150 TreeMap<Integer, ShortenedFunction> shortFunctionMap =
151         new TreeMap<Integer, ShortenedFunction>();
152 TreeMap<Integer, ShortenedIns> shortInsMap =
153         new TreeMap<Integer, ShortenedIns>();
154 List<ShortenedFunction> functions = new ArrayList<ShortenedFunction>();
155 List<ShortenedIns> instructions = new ArrayList<ShortenedIns>();
156
157 for(Map.Entry<Integer, Function> entry : c.getFunctionMap().entrySet()) {
158     ShortenedFunction shortenedFunction = new ShortenedFunction(entry.getValue());
159     shortFunctionMap.put(entry.getKey(), shortenedFunction);
160
161     functions.add(shortenedFunction);
162 }
163
164 for(Map.Entry<Integer, Ins> entry : c.getCodeMap().entrySet()) {
165     ShortenedIns shortenedIns = new ShortenedIns(entry.getValue());
166     shortInsMap.put(entry.getKey(), shortenedIns);
167
168     instructions.add(shortenedIns);
169 }
170
171 binary.setFunctions(functions);
172 binary.SaveFunctions();
173
174 binary.setInstructions(instructions);
175 binary.SaveInstructions();
176
177 p.imports  = Import.GetAllImports(p.getID());
178
179 gui.setLoadedProject(p);
180
181 fillDisplay();
```

Figure 18. Saving functions and instructions in Controller.java.

The properties of the *ShortenedFunction* object includes: a string to represent the virtual address where the function begins, an integer that represents the starting virtual address of the function, an integer that represents the ending virtual address of the function, a list of integers that represent virtual addresses for exit points within the function, a list of integers that represent virtual addresses for entry points within the function and a string that represents an alias for the function. Member functions include getters and setters for these properties.

3.4.5 Updating the graphical user interface. Data persistence is the final stage in the parsing and disassembling of the input file. The framework will use the data from the database to

`

update the graphical display. This contrasts with most frameworks of this type that will use the disassembled or parsed data directly from the binary, not providing a permanent storage mechanism. One framework that does follow a similar model is that of the popular IDA Pro tool, which uses a proprietary file format to store information about a disassembled program (Eagle, 2011). However, the data storage format does not allow for the project concept accomplished here, nor for dynamic searching of multiple assembly mnemonics in a single or across all binaries in a project. At this stage, the user if able to interact with the disassembled program, begin analyzing its functionality and seeking out known/unknown vulnerabilities. The rest of this section discusses the code developed to implement the data displayed in the user interface.

A function called *fillDisplay* is called and updates the primary user interface with function and instruction information from the corresponding data from the database. The function accomplishes this by referencing the properties of the *import* object that contains collections of the *ShortenedFunction* and *ShortenedIns* objects. To support multiple imports per project, the primary graphical interface was designed to use the *TabPane* container from the JavaFX library. This provides a built-in mechanism for the display of tab content and is highlighted in Figure 19.

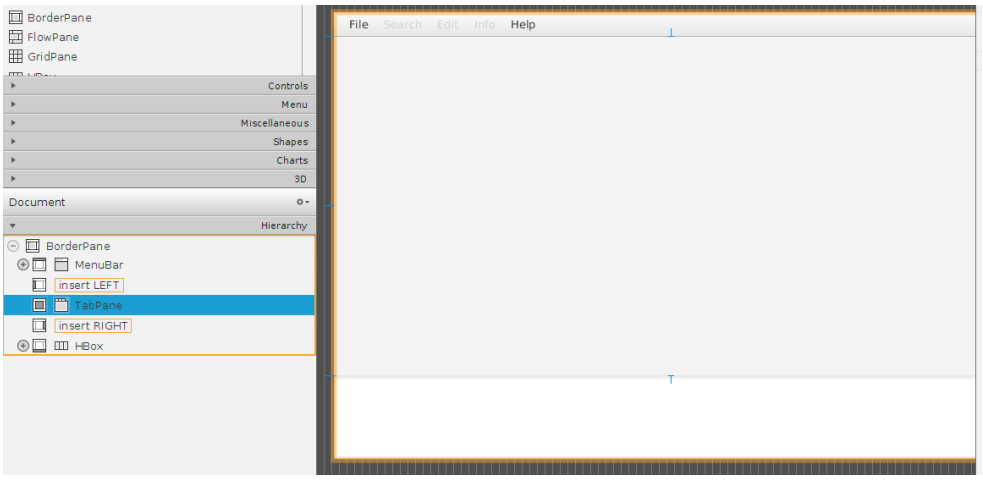


Figure 19. Primary graphical interface highlighting use of the TabPane.

`

The *fillDisplay* function dynamically generates new container objects for each import in the project, a *SplitPane* is utilized to provide function information in the left-hand portion and instruction information in the right-hand portion of each tab. For function display, a *ListView* provides the necessary container. For instruction display a *TableView* is utilized as this provides the ability to bind the appropriate data to columns of the table. The properties from the *ShortenedFunction* and *ShortenedIns* objects are used to populate the *ListView* and *TableView* respectively, then those containers are added to the *SplitPane* object. Once the *SplitPane* has been prepared, a *Tab* object is created for each import and the *SplitPane* added to it as the content. This is performed in a loop, iterating over each import for the project. Once complete, the display is available to the user as seen in Figure 20.

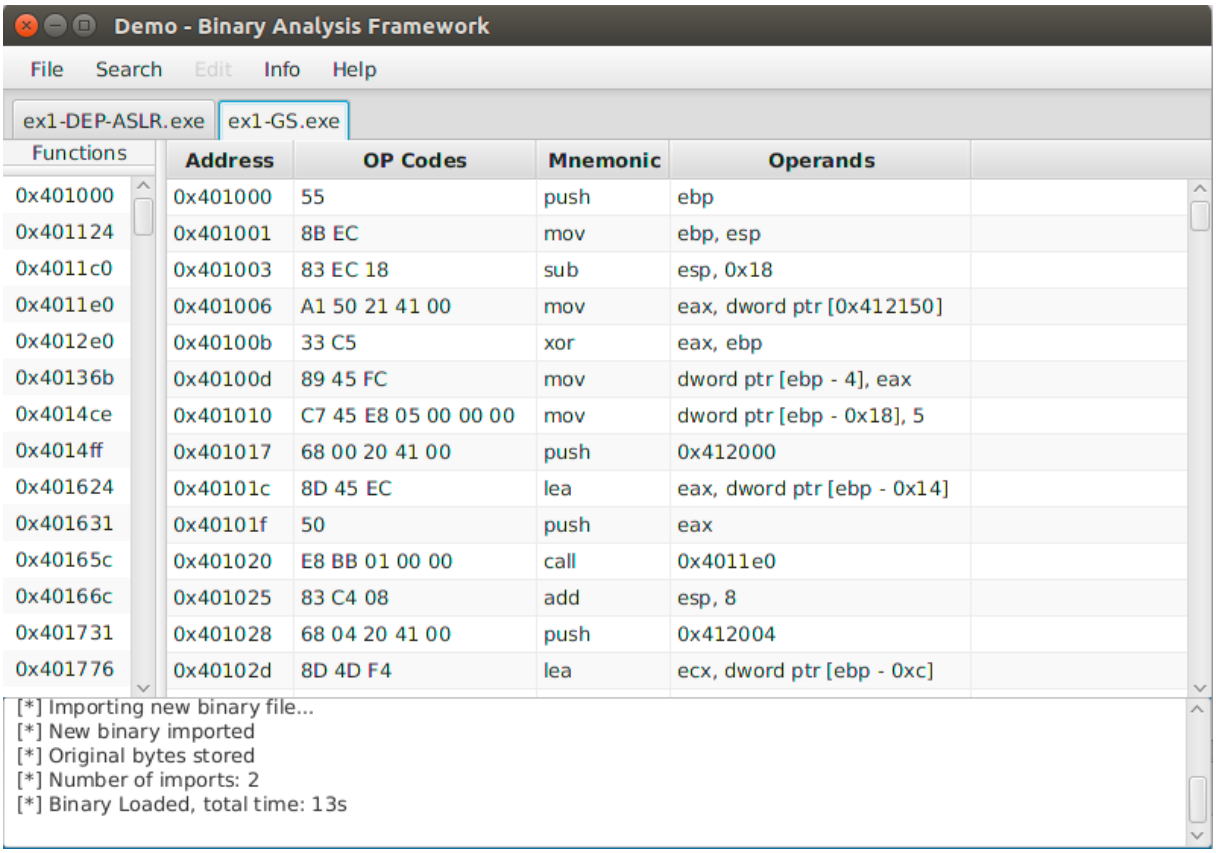


Figure 20. Primary graphical display highlighting use of tabs.

`

3.5 Search Functionality Architecture

Once a project is loaded and there is at least one imported file, the *Search* file menu is enabled. The *Search* file menu exposes four search features: search by mnemonic expression, search by mnemonic sequence, search for return oriented programming (ROP) gadgets and function comparison by mnemonic. The first three search features support a single import and will be performed against the currently selected tab. The final search feature will be performed across all files within the project

Upon selection of the desired menu item, an interface dialog will appear for user interaction. As with the construction of the primary user interface, these dialogs utilize the JavaFX library and FXML for structure. Controllers were also developed to isolate related logic to each interface. However, search functionality leverages the data stored in the database and additional architecture was developed to facility access to this data as well as to maintain the modularity of the framework.

Database access is controlled through the Java classes contained in the *application.database* package. Overall architecture design followed the principles of *N-Tier Architecture*, in which the application was separated into multiple tiers ("N-Tier Data Applications Overview," n.d.). Interactions through the controllers, whether by user or application life-cycle event, went through the data access objects to interface with the database. These objects then return data utilized by the controller classes to populate and update the appropriate user interface element. The data access tier begins with a class *DBConnection,* which implements a single constructor responsible for providing the connection string, username, password and appropriate driver to create a connection with the underlying relational database management system (RDMS). Each class desiring data access can utilize the *DBConnection* class to instantiate a connection

`

object. This work implemented MySQL but it is conceivable that support for additional database systems could be implemented.

Additional classes were created to provide access to the underlying database schema and a class for each table was developed. These classes provide a direct object mapping through properties to the columns of the table they represent and are made available as public members.

3.6 System Design Summary

This chapter provided a formal definition of the Binary Analysis Framework. The components of the framework where presented in detail and the relationships between those components have been defined. This chapter also provided the architecture for the search functionality, its rationale and how it addresses shortfalls in current binary analysis frameworks. Implementation of search functionality is presented in the next chapter.

`

CHAPTER 4

CASE STUDY

This chapter provides results from the development of the Binary Analysis Framework. Four search techniques were developed and will be presented in section 4.1 through 4.4. The ability to identify compile and link time security features was also developed and is discussed in section 4.5. Section 4.6 will present results on analysis of system libraries from Windows 7 32-bit and section 4.7 provides a brief discussion on disassembly output validation.

Of primary interested in developing the Binary Analysis Framework was in exploring a novel way to search binary files (i.e. the input files), using regular string patterns in the form of assembly instructions. The string inputs would be compared directly to the disassembled output that was stored in the database using string comparison functions. This work experimented with where the string comparison would be made, in the application code (i.e. Java), in the structured query language (SQL) at the database or a combination of both techniques. The framework also implemented search capabilities that expand beyond a single input file. This work experimented with expanding the search capability to allow for search and comparison across all files in a project. The following search functions were implemented using the framework developed: search by mnemonic expression, search by mnemonic sequence, search for return oriented programming (ROP) gadgets and function comparison search by mnemonic. In addition to search capabilities,

`

the framework also developed ways in which security information could be identified, such as compiler and linker options.

The results discussed here show that an effective framework can be constructed utilizing a relational database for data persistence. Disassembly output can be efficiently stored in a normalized database schema and used to provide both output for analysis and novel search techniques. Search techniques are introduced in the following sections of this chapter.

4.1 Search by Mnemonic Expression

Search by mnemonic expression allows the user to perform a search utilizing the syntax of assembly instructions. This technique allows the user to search program logic within the target binary file to perform program analysis. This technique employs special patterns not part of normal assembly syntax. This enables much broader search patterns and facilitates greater exploration of program functionality.

Search by mnemonic expression allows a user to input a search pattern in the form of an assembly instruction, which is a mnemonic followed by any necessary operands. To assist in providing a standard delimiter when parsing the mnemonic and operands by the framework, a hashtag (‘#’) character is used to delimit the segments of the input search sequence. Figure 21 depicts an example search for the instruction ‘*call eax’*, in which the mnemonic is the segment *call*

`

and the operand is the segment *eax* (Intel 64 and IA-32 Architectures Software Developer's Manual, n.d.).

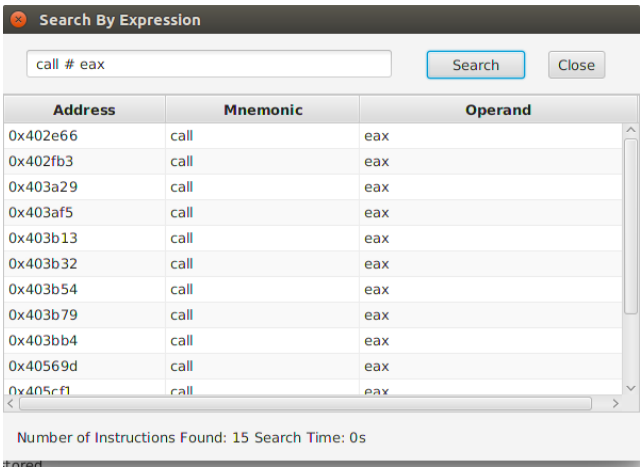


Figure 21. Example search by expression without use of special patterns.

4.1.1 Components of the mnemonic expression search. This interface utilizes the *MnemonicSearch.fxml* document and was assigned the *SearchController* class to handle all user interaction. Upon entering the desired search sequence, the user clicks the *Search* button. This button corresponds to the *searchByMnemonicExpression* function defined in the controller class. As discussed in the modularity of the framework, controller functions are responsible for obtaining the appropriate results from the database and updating the user interface. To achieve this, the static method *searchByMnemonicExpression* from the *Import* class is invoked. The *Import* class was expanded to become the primary interface object between user interface elements, controllers and the database. The function *searchByMnemonicExpression* returns a list of *ShortendInsn* objects which are used to populate the *TableView* object in the search results interface. This method constructs an SQL query to identify matching instructions. The base of the query is defined as a static string in the class definition and contains the *SELECT* portion of the SQL statement. This

query utilizes the *Instruction* table to obtain the desired result set. This method then dynamically constructs elements of the *WHERE* clause to provide the search pattern as desired by the user.

Figure 22 depicts the function *Import.searchByMnemonicExpression.* This function was declared as static and requires two arguments: the id of the import to search and a string value representing the search input. The import id is required as argument one due to the static definition of the function, this value provides the primary key to identify the import to search. The second argument provides the raw search string the user entered in the textbox in the user interface. The function begins by instantiating the *DBConnection* class on line 377 in Figure 22, which provides a connection to the database server. On line 379 the search sequence is split by the hashtag delimiter and each part stored as an element of a string array. The next function that is called is *buildMnemonicExpression*, which is a static method defined in the *Import* class. This method requires a single argument, an array of strings which represent the parts of the search sequence. This method is called on line 381, immediately after the user input is split.

```
374     DBConnection _db = null;
375
376     try {
377         _db = new DBConnection();
378
379         String[] query = exp.split("#");
380
381         String operands = buildMnemonicExpression(query);
382
383         java.sql.PreparedStatement prepared_stmt =
384                 _db.conn.prepareStatement(search_mnemonic + operands);
385         prepared_stmt.setString(1, query[0]);
386         prepared_stmt.setInt(2, importID);
387
388         ResultSet rs = prepared_stmt.executeQuery();
389
390         while(rs.next()) {
391             results.add(new ShortenedIns(
392                     rs.getString("opcodes"),rs.getInt("address"),
393                     rs.getString("mnemonic"), rs.getString("operands"),
394                     rs.getInt("opsize")));
395         }
```

Figure 22. Function Import.searchByMnemonicExpression from Import.java.

`

4.1.2 Function buildMnemonicExpression. The function *buildMnemonicExpression* is used by all search functions to standardize the format of the user input. The function does this by building the *WHERE* clause of the SQL query used to search the *Instructions* table by leveraging regular expressions and the construction of the regular expression syntax. This function also allows the user to utilize special search patterns in their sequence, broadening the number of instructions that can potentially match in the user's input. The following special input strings will be recognized by the framework: *imm, r32* and *mem*. The *imm* pattern represents an immediate value and is defined by Intel as "data encoded in the instruction itself as a source operand" (*Intel 64 and IA-32 Architectures Software Developer's Manual*, n.d.). An example of a query supported by this pattern would be '*push imm'*, in which the search would find any instruction with a *push* mnemonic and any immediate value as the operand. In contrast, searching without this pattern would require the user to know the value of the operand that they wanted to search for, such as *push 0*. Figure 23 highlights the use of this special search pattern.

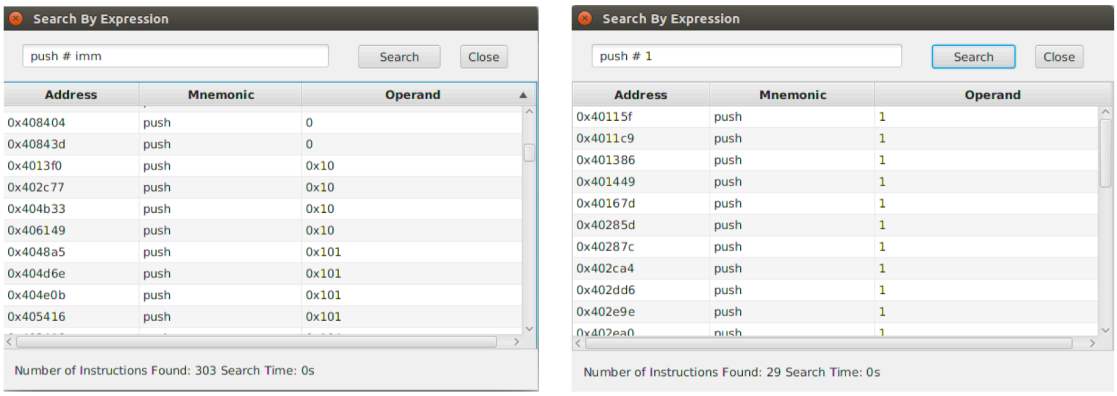


Figure 23. Example search using special instruction imm.

The pattern *r32* represents one of the 32-bit registers available, such as *EAX*. This pattern provides similar support as that described in the previous pattern, it allows the user to search instructions without specifying the exact register they wish to search for. Figure 24 highlights the use of this special pattern.

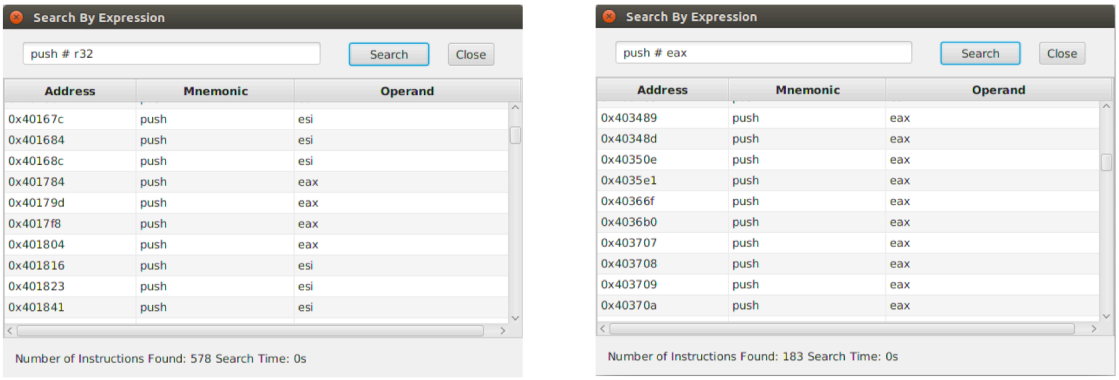


Figure 24. Example search using special instruction r32.

Finally, *mem* represents access to a memory location, such as offsets relative to the EBP or ESP register or direct memory locations prefixed by '0x', which indicates that virtual address in hexadecimal notation. The special patterns increase the user's ability to explore instructions within the file by broadening the number of potential matches.

The function *buildMnemonicExpression* iterates over each part of the string array argument beginning at line 338 in Figure 25, testing for the presence of one of the special search patterns.

`

```
330 public static String buildMnemonicExpression(String[] query) {
331
332     String operands = "";
333
334     if (query.length > 1) {
335
336         operands += " AND operands REGEXP '.*";
337
338         for(int i = 1; i < query.length; i++) {
339
340             String op = query[i].trim().replace("$", "");
341
342             if(op.equals("imm"))
343             {
344                 operands += "((0[xX][0-9a-fA-F]+)|([0-9]{1,}))";
345             }
346             else if(op.equals("r32"))
347             {
348                 operands += "[xi]{1}";
349             }
350             else if(op.equals("mem"))
351             {
352                 operands += "(ebp|esp|0x).*";
353             }
354             else
355             {
356                 operands += op;
357             }
358
359             if ( i < query.length - 1 )
360             {
361                 operands += ",.*";
362             }
363         }
364         operands += "$'";
365
366     }
367     return operands;
368 }
```

Figure 25. Function Import.buildMnemonicExpression from Import.java.

Regular expression syntax has been defined for each case and can be seen in line 344 for the value *imm*, line 348 for the value *r32* and line 352 for the value *mem*. If the search segments do not match any special case, then a direct comparison in the SQL is made and regular expressions are not utilized. Each iteration of the loop causes the results of the comparisons to be concatenated to a single string object, which is then returned upon completion of the function. This value is concatenated with the base query in line 381 in Figure 22 and used to instantiate a prepared

`

statement object. Once instantiated, parameter values are assigned and the SQL executed. Results are iterated and used to create *ShortenedInsn* objects and appended to a list collection, which is returned to the calling controller method and used to populate the user interface.

4.2 Search by Mnemonic Sequence

Search by Mnemonic Sequence extends the logic developed in the previous search technique by expanding the search functionality to include the ability to search by multiple sequences of sequential instructions. Program analysis frequently requires the identification and analysis not of single instructions, but sequences of instructions that lead to a specific program state. The quicker a researcher can identify these patterns, the more effective their time is spent in performing analysis. This search technique allows users to search by pattern instead of individual instructions, enabling them to quickly identify areas of a program that may be of interest for deeper analysis.

Search by Mnemonic Sequence utilizes the concept of the search expression being an instruction sequence, but expands the search functionality to include the ability to search by multiple sequences of instructions. Figure 26 represents an example search, in which the user is searching for the instruction '*push r32'* followed by a second instruction '*pop r32'*.

4.2.1 Special search pattern identifiers. When performing this search, any matches for the first sequence must also be followed by the second sequence. This search functionality further expands upon the special patterns and allows for the use of the dollar-sign character ('$'). When added to a special pattern, it uses the corresponding matching operand to restrict matches in later comparisons. An example of this feature can be seen in Figure 26, the dollar-sign was prepended to the special pattern *r32*. During the search, if a pattern is matched it will be used as a value in

`

subsequent instructions. In this example, the first instruction matched on the register *EAX*. This value was used in the second instruction instead of matching any register, which is the behavior of the special pattern *r32* without the dollar-sign prefix.

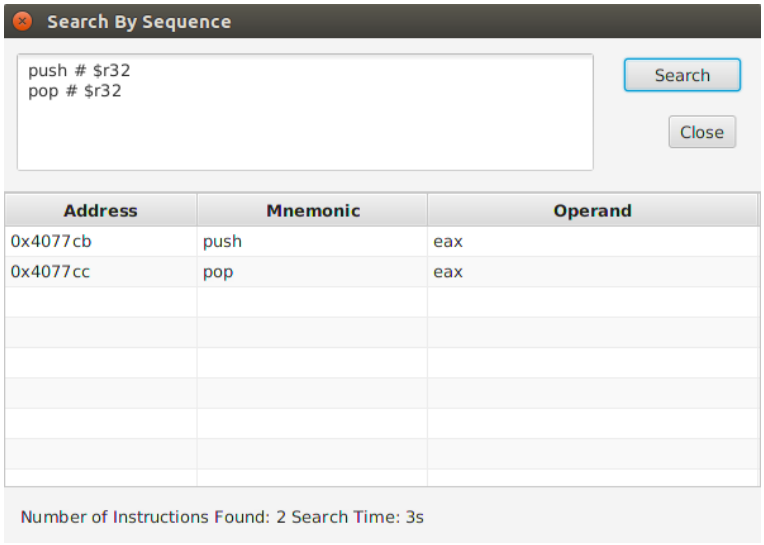


Figure 26. Example search by mnemonic sequence.

4.2.2 Components of the mnemonic sequence search. This feature is supported by a user interface element, *MnemonicSequenceSearch.fxml,* and supporting controller, *SearchSequenceController.* Once the user clicks the *Search* button, the method *searchByMnemonicSequence* is called in the controller. This method calls the static method *serachByMnemonicSequence* from the *Import* class. This function returns a list of *ShortendInsn* objects which is used to populate the *TableView* in the search results interface. This method constructs an SQL query to identify matching instructions. The base of the query is defined as a static string as part of the class definition. This method then dynamically constructs elements of the *WHERE* clause to provide the search pattern as desired by the user.

Figure 27 and Figure 28 depict the function *Import.searchByMnemonicSequence.* This function begins by instantiating a *DBConnection* object, then splitting the user input by carriage-

`

return and line-feed as seen at line 415. This is done to store each individual search expression in a string array. To begin searching, the first expression is then split by hashtag delimiter and the *WHERE* clause of the SQL is constructed, this is done using the *buildMnemonicExpression* function.

```
413     DBConnection _db = null;
414
415     String[] expressions = exp.split("\\r\\n|\\n|\\r");
416     int num_sequences = expressions.length;
417     int next_address = 0;
418
419     String[] query = expressions[0].split("#");
420     String operands = buildMnemonicExpression(query);
421
422     try {
423         _db = new DBConnection();
424
425         java.sql.PreparedStatement prepared_stmt =
426                 _db.conn.prepareStatement(search_mnemonic + operands);
427         prepared_stmt.setString(1, query[0]);
428         prepared_stmt.setInt(2, importID);
429
430         ResultSet rs = prepared_stmt.executeQuery();
431         List<ShortenedIns> first_ins = new ArrayList<ShortenedIns>();
432
433         while(rs.next())
434         {
435             first_ins.add(new ShortenedIns(rs.getString("opcodes"),
436                     rs.getInt("address"), rs.getString("mnemonic"),
437                     rs.getString("operands"),rs.getInt("opsize")));
438         }
439
440         rs.close();
```

Figure 27. Function Import.searchByMnemonicSequence from Import.java.

The result from returned from the function *buildMnemonicExpression* is concatenated with the base SQL query and used to search for matching instructions in the *Instruction* table. Results are iterated and used to create *ShortenedInsn* objects and added to a list collection.

`

```
442         for(int z = 0; z < first_ins.size(); z++)
443         {
444             ShortenedIns tmpIns = first_ins.get(z);
445             next_address = tmpIns.getAddress() + tmpIns.getOpSize();
446 
447             tmp.add(tmpIns);
448 
449             innerloop:
450             for(int x = 1; x < expressions.length; x++)
451             {
452 
453                 String[] sub_query = expressions[x].split("#");
454                 String[] firstInsOperands = tmpIns.getOperands().split(",");
455 
456                 if(sub_query[1].contains("$"))
457                 {
458                     if(query[1].contains("$"))
459                     {
460                         sub_query[1] = firstInsOperands[0];
461                     }
462                     else if(query[2].contains("$"))
463                     {
464                         sub_query[1] = firstInsOperands[1];
465                     }
466                 }
467                 else if (sub_query[2].contains("$"))
468                 {
469                     if(query[1].contains("$"))
470                     {
471                         sub_query[2] = firstInsOperands[0];
472                     }
473                     else if(query[2].contains("$"))
474                     {
475                         sub_query[2] = firstInsOperands[1];
476                     }
477                 }
```

Figure 28. Function Import.searchByMnemonicSequence from Import.java.

With the initial collection of results, the next stage of the function is to search for additional sequences of instructions, if provided in the users search input. A loop starting at line 442 is created to iterate over the collection of search results obtained from the first instruction. To determine the next instruction to search, the address of the current instruction is obtained and the size of the instruction, in bytes, is added. The resulting virtual address is the location of the next instruction and used in the SQL query to identify and search the correct sequential instruction.

`

This is possible since each record in the *Instruction* table has an address column, which is the starting virtual address of the instruction. This is accomplished at line 445.

The next control structure is a FOR loop created at line 450, which iterates over the remaining search expressions. Inside this loop the search expression is split by hashtag character and stored in a string array. Before the *WHERE* clause for the SQL statement is generated, the operands are checked for the existence of the dollar-sign prefix. A series of conditional statements was implemented to provide this functionality and can be seen starting at line 456. If either operand contains the dollar-sign prefix, then the value used for this part of the search pattern will be the corresponding result from the initial match. If not, then the operand will be left unchanged. After determination of the dollar-sign prefix, the function calls *buildMnemonicExpression* and uses the returned string value to concatenate to the base SQL query. The result set returned by the SQL query are iterated and added to a temporary collection of *ShortenedInsn*. This process continues recursively if there are additional instructions to search for; calculating the next instructions virtual address, testing for the dollar-sign prefix, building the *WHERE* clause SQL and iterating the result set. Once all instructions have been searched, the function returns the collection of *ShortenedInsn* objects and the invoking controller method updates the user interface appropriately.

4.3 Search for Return-Oriented Programming (ROP) Gadgets

The ability to search for ROP gadgets deviates slightly from the previously described functionality. This feature is supported by a user interface element, *ROPSearch.fxml,* and controller, *ROPSearchController.* Once the user clicks the *Search* button, the method *searchForROPGadgets* is called in the controller. This method calls the static method

`

searchROPGadgets from the *Import* class. This function returns a list of *ShortendInsn* objects which is used to populate the *TableView* in the search results interface.

Figure 30 depicts the function *Import.searchROPGadgets*. In the user interface (Figure 29), the user supplies the mnemonic instruction in which they want to enumerate gadgets for, this value being used as the ending instruction. To obtain a list of all instructions in the imported file, a call to *searchByMnemonicExpression* at line 568 is performed. As previously discussed, this method returns a collection of *ShortendInsn* objects. This collection of *ShortenedInsn* objects is iterated and the virtual address, provided by the address property, is used to calculate the address five bytes higher at line 579.

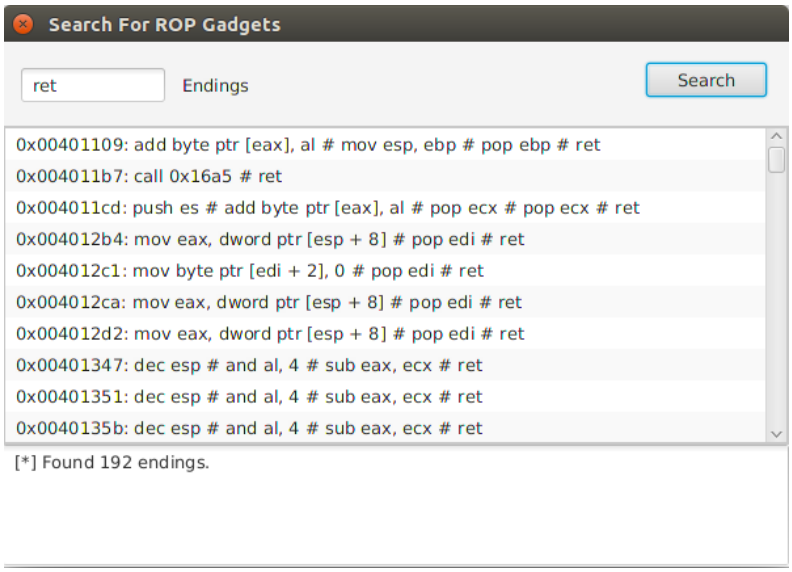


Figure 29. Example search for return-oriented programming gadgets.

```
568     ObservableList<ShortenedIns> obsInstructions =
569             FXCollections.observableArrayList(
570             Import.searchByMnemonicExpression(importID,endingsToSearch));
571
572     String[] opcodes = selected_import.code_section.content.split(" ");
573
574     for (int i = 0; i < obsInstructions.size(); i++) {
575
576         ShortenedIns si = obsInstructions.get(i);
577
578         int offset = si.getAddress() - 0x401000;
579         int adjusted_offset = offset - depth;
580
581         List<Byte> bytes = new ArrayList<Byte>();
582
583         for(int j = adjusted_offset; j <= offset; j++) {
584
585             byte value = (byte) ((Character.digit(opcodes[j].charAt(0), 16) << 4)
586                     + Character.digit(opcodes[j].charAt(1), 16));
587
588             bytes.add(value);
589         }
590
591         byte[] code = new byte[bytes.size()];
592
593         for(int x = 0; x < bytes.size(); x++)
594         {
595             code[x] = bytes.get(x).byteValue();
596         }
597
598         try{
599             String tmp_gadget = "";
600
601             Capstone cs = new Capstone(Capstone.CS_ARCH_X86, Capstone.CS_MODE_32);
602
603             Capstone.CsInsn[] allInsn = cs.disasm(code, offset - depth);
604
605             for (int z = 0; z<allInsn.length; z++)
606             {
607                 tmp_gadget += allInsn[z].mnemonic + " " + allInsn[z].opStr + " # ";
608             }
609
```

Figure 30. Function Import.searchROPGadgets from Import.java.

4.3.1 Calculating offsets from virtual addresses. Since the framework does not provide virtual address emulation, virtual addresses must be converted to raw offsets from the beginning of the code section of the original import. The code section bytes are stored during initial parsing and disassembly and retrieved here for searching and further disassembly. The *Import* class

defines a member *code_section*, which is an object of type *section.* This object represents the *section* table and provides access to the raw bytes of the code section of the imported file.

At line 572 the member function *content* is called on the *section* object and the raw bytes are returned and stored in a local string array, *opcodes.* The loop at line 583 is responsible for iterating each of the characters in this string array and converting them to the corresponding byte value, which is saved in a byte array named *bytes.* Once an offset has been calculated, the appropriate index can be referenced in this byte array and the opcodes disassembled. The code responsible for disassembly begins at line 599 by instantiating an instance of the Capstone library. This object is then used to disassemble the appropriate bytes from the byte array and produces *CsInsn* objects, which provide access to each instructions mnemonic and operands. Each instruction is concatenated to a string object as returned as a singular ROP gadget. This process repeats for each original instruction that was found in the imported files disassembly.

4.4 Simultaneous Search

Simultaneous search enables users to compare functions across all imported files in a project. This search technique allows users to identify functionality that is similar in arbitrary binary files. This is accomplished by utilizing all identified functions in an import, performing a mnemonic comparison with each instruction. Current search methods provide a percentage of program similarity by performing a comparison at the binary level. While this helps to identify related files, it does not provide any insight into what program logic is similar. Simultaneous search not only provides similarity results, but also allows the user to investigate precisely what program logic is similar.

`

For each function, the instructions are retrieved from the database and the instruction mnemonic used for comparison. If a function contains an identical sequence of mnemonics, then it is determined to be a matching function. This process is performed for each function across all imports in the project. An example of the utility of this feature is in analyzing malware, which often makes minor modifications to the application to change the resulting byte pattern of the compiled program. This is done to disrupt signature based detection without the need to significantly change the code base of the program (Sikorski & Honig, 2012). When analyzing unknown programs, this feature can be used to identify similar functionality, reducing the time the analyst spends when piecing together the functionality of a program. Figure 31 shows the results of this search feature, the starting virtual address for each matching function is displayed.

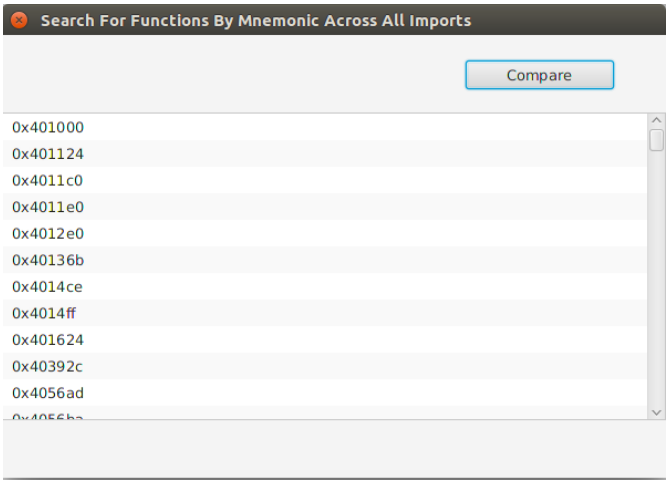


Figure 31. Example Function Comparison Search

4.4.1 Components of function comparison search. This feature is supported by a user interface element, *FunctionSearch.fxml,* and controller, *FunctionSearchController.* Once the user clicks the *Search* button, the method *searchForFunctions* is called in the controller. This method calls the static method *searchFunctionsAcrossImports* from the *Import* class. The function

searchFunctionsAcrossImports returns a list of *ShortenedFunction* objects which is used to populate the *TableView* in the search results interface.

Figure 32 depicts the function *searchFunctionsAcrossImports*. This function begins by using the argument *importID* to instantiate an import object, this represents the import from the currently selected tab and serves as a starting point to begin the search, the functions within this import will be used to search within other imports.

```
630         Import im = new Import(importID);
631
632         List<ShortenedFunction> funcs = im.getFunctions();
633         List<Import> projectImports = p.imports;
634         List<ShortenedFunction> matchingFunctions =
635                 new ArrayList<ShortenedFunction>();
636
637         for (int j = 0; j < projectImports.size(); j++ )
638         {
639             Import tmpImport = projectImports.get(j);
640
641             if(im.getID() != tmpImport.getID())
642             {
643                 for ( int i = 0; i < funcs.size(); i++)
644                 {
645                     List<ShortenedIns> tmpListIns =
646                             im.getFunctionInstructions(funcs.get(i));
647
648                     List<ShortenedFunction> tmpResults =
649                             searchImport(tmpListIns, tmpImport);
650
651                     if (tmpResults.size() > 0)
652                     {
653                         matchingFunctions.addAll(tmpResults);
654                     }
655                 }
656             }
657         }
```

Figure 32. Function searchFunctionsAcrossImports from Import.java.

At line 632 the functions for the current import are stored in a list collection. At line 633 another list collection is created and assigned all the imports for the current project, this allows for the search functionality to compare functions across all imports. The final list collection, at line 634, is created to store the results of any matching functions and returned to the invoking controller

`

method, which will then be used to update the user interface (Figure 31). The loop structure at line 637 iterates all the imports in the project. If the project that is currently selected through the iteration of the loop does not match the originally selected import, then another loop will begin iterating each function from the original import, this is defined at line 643. Two temporary list collections are created: one contains all of instructions for the current function, the other will contain the results returned from the function call to *searchImport.*

4.4.2 Function searchImport. Figure 33 depicts the function *searchImport.* This function expects two arguments: the first is a list collection of instructions to search for, the second is the import to be searched.

```
664     List<ShortenedFunction> results = new ArrayList<ShortenedFunction>();
665     List<ShortenedFunction> funcs = im.getFunctions();
666
667     for(int i = 0; i < funcs.size(); i++)
668     {
669
670         List<ShortenedIns> tmpListIns =
671                 im.getFunctionInstructions(funcs.get(i));
672
673         int seek_len = tmpListIns.size();
674
675         if (sourceFuncIns.size() == tmpListIns.size())
676         {
677             int j = 0;
678
679             while(j < seek_len
680                     && tmpListIns.get(j).getMnemonic().
681                         equals(sourceFuncIns.get(j).getMnemonic()))
682             {
683                 j++;
684
685                 if(j == seek_len)
686                 {
687                     results.add(funcs.get(i));
688                 }
689             }
690         }
691     }
692     return results;
```

Figure 33. Function searchImport from Import.java.

The outer for loop defined at line 667 iterates each function from the target import. Inside the loop, the instructions for the target function are retrieved from the database by a call to *getFunctionInstructions*, which is a member function of the *Import* class. The function *getFunctionInstructions* returns a list collection of *ShortenedInsn*, which will be compared against the objects in the first argument. A while loop at line 679 provides the control structure to begin mnemonic comparison, each instruction from both collections is compared in sequential order by mnemonic. If all the mnemonics match, then the *ShortenedFunction* object is added to the result set. If one instruction does not match than the loop breaks and the next function is compared. This process continues until all imports, and their functions, are compared to the originally selected import.

4.5 Security Feature Identification

The identification of security features enabled in an imported program supported by the framework includes data execution prevention, address space layout randomization, control flow guard and stack guard. These features are determined at compile or link time during executable program construction. This information is identified by the framework during the parsing of the file format and stored in the *Characteristics* table, with a bit field defined for each property. Once a project is loaded, a file menu option *Info* is enabled, this provides a menu item *Security Info* which opens a user interface element.

4.5.1 Components of security feature identification.

To support identification of this information, the *Code* object (as discussed in the Binary Parsing section) implements four Boolean properties: *depEnabled*, *aslrEnabled*, *cfgEnabled* and *stackGuardEnabled*. These properties are then set per functions implemented in the sub-classes. This work focused on the parsing of the PE

`

file format, the corresponding class would be the *PE* class. Within the PE file format, this information is determined from the word member *DllCharacteristics* of the *IMAGE_OPTIONAL_HEADER* structure ("IMAGE_OPTIONAL_HEADER structure," n.d.). To identify the use of ASLR, DEP and CFG, three bit masks were defined as private members in the *PE* class:

- IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE
- IMAGE_DLLCHARACTERISTICS_NX_COMPAT
- IMAGE_DLLCHARACTERISTICS_GUARD_CF

The values assigned to these private members were 0x0040, 0x0100 and 0x4000, respectively. The static members were then used to perform a bitwise AND comparison to the word member *DllCharacteristics* of the imported file. If the results of the comparison were a non-zero, positive value, then the appropriate Boolean member is set to true, indicating the use of that feature in the import. The default state of the Boolean field is false, and does not require updating if the results of the comparison is false. Figure 34 depicts the process of identification through the function *isDEPEnabled* and is representative of the process of detecting the use of CFG and ASLR.

```
64          public Boolean isDEPEnabled() {
65              if((characteristics & IMAGE_DLLCHARACTERISTICS_NX_COMPAT) != 0)
66                  return true;
67
68              return false;
69          }
```

Figure 34. Function isDEPEnabled from PE.java.

4.5.2 Identification of stack guard. Detecting the use of the stack guard required a different approach. The detection of the use of the stack guard focused on the implementation

`

used by Microsoft and the resulting Microsoft Visual C/C++ compiler. The framework allows for the expansion of this feature to identify other techniques used to add stack guard support. If a binary supports the stack guard, then a pointer is located at an offset of +0x3C from the base of the ".rdata" section, what the pointer points to is updated at runtime and is the dynamically generated stack guard value. The absence of a pointer value at this offset indicates that the binary is not using the stack guard feature. The framework locates this offset and accesses the integer value at that location. If the value is not zero, then the import is determined to have utilized the stack guard. Figure 35 depicts the function *isStackGuardEnabled.*

```
89  public Boolean isStackGuardEnabled() {
90
91      ImageDataDirectory loadConfig =
92              fileHeader.getImageOptionalHeader().getDataDirectory()[10];
93      ImageSectionHeader rdataSegmentHeader =
94              fileHeader.getSegment(".rdata");
95
96      if (rdataSegmentHeader != null) {
97
98          int dataOffset =
99                  (int) (loadConfig.getAddress()
100                         - rdataSegmentHeader.getVirtualAddress());
101
102         int foa = (int) rdataSegmentHeader.getRawDataPosition() + dataOffset;
103
104         if(LittleEndian.getUInt(bytes, foa+0x3C) > 0){
105             return true;
106         }
107     }
108     return false;
109 }
```

Figure 35. Function isStackGuardEnabled from PE.java.

4.6 System Analysis

The Binary Analysis Framework was evaluated for performance against several key Windows libraries. The libraries were obtained from an installation of Windows 7 Service Pack 1 32-bit. The results, along with the libraries evaluated, are reported in Table 3. The information includes: library name, file size, total processing time in seconds and whether the library utilizes control-flow guard, stack guard, ASLR and DEP. It is worth noting that control-

flow guard was not implemented for Windows 7 system libraries. No issues were encountered during parsing and security feature identification of the analyzed system libraries.

4.7 Disassembly Verification

Spot check validation was performed on the disassembly output generated by the Binary Analysis Framework. IDA Pro provides an export feature that allows a user to create a disassembly listing of any function in the database. The format of this export is section name colon virtual address, mnemonic and operands. Figure 36 depicts an example listing.

```
.text:00401000 push    ebp
.text:00401001 mov     ebp, esp
.text:00401003 sub     esp, 18h
.text:00401006 mov     eax, ___security_cookie
.text:0040100B xor     eax, ebp
.text:0040100D mov     [ebp+var_4], eax
.text:00401010 mov     [ebp+var_18], 5
.text:00401017 push    offset aOne     ; "one"
.text:0040101C lea     eax, [ebp+var_14]
.text:0040101F push    eax             ; char *
.text:00401020 call    _strcpy
.text:00401025 add     esp, 8
.text:00401028 push    offset aTwo     ; "two"
```

Figure 36. Listing Generated by IDA Pro

The listing output provided instruction information that could be compared with similar output generated by the framework and the results of the comparison used to determine accuracy. A Python script was developed that parsed this information to isolate the mnemonic, it then used the sequence of mnemonics for comparison in an import within a project. The script would query all instructions for a given import and begin comparing to those provided from the listing file. If all the instruction mnemonics matched, the starting address of the sequence of instructions is printed to standard out. This can be verified with the function address used to generate the listing

`

file. If the entirety of the sequence of instructions does not match, then no results are displayed. The results of comparing the function at virtual address 0x00401000 from the listing file generated by IDA Pro and the same file imported by the framework is demonstrated in Figure 37.

```
user@ubuntu:~/Desktop/Research$ python test_listing.py -f ex1-DEP-ASLR-Main.lst
-p Demo -i ex1-DEP-ASLR.exe
[*] Beginning analysis on ex1-DEP-ASLR-Main.lst on Demo --> ex1-DEP-ASLR.exe
[!!] Found Match: 0x00401000
```

Figure 37. Output of Disassembly Verification

4.8 Case Study Summary

This chapter presented results from the development of the Binary Analysis Framework. Search functionality was the primary contribution of this work and four novel search techniques were developed using the framework. The final chapter will detail what was learned during the development of the Binary Analysis Framework, future work and summarize contributions to the study of reverse engineering.

`

CHAPTER 5

CONCLUSION

The Binary Analysis Framework allows security researchers to perform full program analysis from software in a binary state. This work introduced a novel search architecture for binary analysis that utilized a relational database. In addition, functionality was developed to identify security information supported by the intended operating system of the software. Interaction with the framework is facilitated through a graphical user interface, which aids the researcher in identifying, confirming and exploring security issues in the target software.

5.1 Contributions

A data persistence architecture was developed using an open-source database system. The benefits of this design are:

- Reduced time in performing program analysis
- An architecture for grouping related binaries
- A novel search architecture based on disassembly output

5.1.1 Reduced time in performing program analysis. The development of a relational database for data persistence allows for the program to load data from programs already disassembled directly from the database. This contrasts with having to perform the disassembly of the input file each time program analysis is desired.

5.1.2 Architecture for grouping related binaries. Related input files can easily by grouped together in a project construct. This increases the efficiency of analysis as all related input

`

files can be accessed from a singular graphical user interface. Current frameworks require an instance of the analysis program to be open per file, increasing time requirements in switching between these instances.

5.1.3 Search architecture based on disassembly output. The search architecture allows for searching by assembly instruction (or disassembly output) and utilized string comparison functions, regular expressions and database language features to perform search comparison. Three novel search methods were developed from this architecture: search by mnemonic expression, search by mnemonic sequence and simultaneous search.

Search by mnemonic expression allows a user to input a search pattern in the form of an assembly instruction. This allows the user to search arbitrary instructions within the target binary file to explore program logic and perform program analysis. This search feature is enhanced by the development of special patterns, which enable much broader search patterns and facilitates greater exploration of program functionality.

Search by mnemonic sequence builds upon the logic developed in the mnemonic expression search by utilizing the concept of the search expression being an instruction sequence, but expands the search functionality to include the ability to search by multiple sequences of sequential instructions. Mistakes in software, to include those that lead to security vulnerabilities, are often the result of multiple sequences of assembly instructions. Some of these instructions are well known, while others need to be identified on a per case basis. This feature allows users to search by pattern instead of individual instructions, enabling them to quickly identify areas of a program that may be of interest for deeper analysis.

Simultaneous search enables users to perform cross-binary analysis. This search feature allows users to identify functionality that is similar in arbitrary binary files. It does this by

`

comparing all instruction mnemonics per function within all imports in a project, allowing the user to identify similar program logic. Current search methods provide a percentage of program similarity by performing a comparison at the binary level. While this helps to identify related files, it does not provide any insight into what program logic is similar. Simultaneous search allows the user to investigate precisely what programming logic is similar.

5.2 Lessons learned.

5.2.1 Unexpected detection of stack guard value.

During research detecting the use of the stack guard, an unanticipated behavior was observed. When compiling a program, the user can use the "/GS" compiler option to enable the use of the stack guard and the "/GS-" to disable it ("/GS (Buffer Security Check)," n.d.). Microsoft defines the use of the security guard as a value added to the beginning of a function "that the compiler recognizes as subject to buffer overrun problems" ("/GS (Buffer Security Check)," n.d.). This generally means that any user defined functions that contain a buffer of any type will have the stack guard added to detect a buffer overrun condition. Sample programs were constructed that utilized buffers and the stack guard and was successfully detected by the logic implemented in the framework. However, when the stack guard was disabled using the "/GS-" option, the use of the stack guard was still observed in the sample programs. Upon further investigation, the library functions in the program were identified to be using the stack guard. This is important to note, as the framework may identify the use of the stack guard in library code even when user defined code is not utilizing it. In that case, a security researcher would still want to investigate the program for potential buffer overflow sites.

`

5.3 Future Work

The Binary Analysis Framework was designed with modularity in mind, which provides for the expansion of the framework in several key areas.

5.3.1 Framework expansion. The number of file formats and architectures supported by the framework can be expanded to those supported by the Capstone library, which is responsible for the disassembly of the input file. Capstone is an independently maintained project with broad community support. Expansion of the Capstone project will also impact the Binary Analysis Framework, as the framework will benefit from any future expansion of the Capstone project in the number of file formats and architectures it can support. The expansion of the number of file formats and architectures supported by the Binary Analysis Framework will provide utility to a broader number in the research community.

5.3.2 Identification of code obfuscation. Reverse engineering is a core activity when analyzing malicious software, or malware (Sikorski & Honig, 2012). To complicate the process of analysis, malware authors routinely use code obfuscation techniques, complicating the process of disassembly. Malware authors will additionally use malformations in the file format with the goal of disrupting the ability to parse the file format correctly (Hahn & Register, 2014). The Binary Analysis Framework does not attempt to detect such techniques and could be expanded in this area, providing for more robust detection of protected or malformed files. Detecting such files provides better information to be used during disassembly to help ensure accurate results are displayed to the researcher.

5.3.3 Expansion of search capabilities. The Binary Analysis Framework developed search techniques involving the disassembled instructions stored in a relational database. This architecture creates novel ways of performing assembly-level searches using structured query

`

language, string functions and regular expressions. Search capabilities are currently supported in two forms: on a single import or across all imports in a project. However, current search functionality across all imports is limited to function-level comparison. The single-import search functionality can be expanded to include all imports in a project. Additional search techniques not considered in this work can also be explored.

5.3.4 User experience. Finally, the graphical user interface can be expanded to provide a more fluid experience of navigating and exploring the disassembly output. Call graphs are not used and the user has no ability to navigate functions using hyperlinks. Current user interface design requires the user to scroll linearly through the disassembly output. Function and operand renaming could also be introduced, which would enhance the analysis performed by the researcher by allowing them to add clarity and meaning to the disassembly output.

5.4 Limitations

The Binary Analysis Framework may not be suitable for all binary files. The following are significant limitations that should be evaluated when considering the adoption or expansion of this work.

5.4.1 Detection of code obfuscation. Code obfuscation, file format malformations and other anti-analysis techniques are often employed by malicious actors to avoid detection and complicate the process of analysis. These techniques can also be employed by non-malicious entities to provide protection of intellectual property or inhibit the discovery of potential software vulnerabilities. Regardless of the desired outcome, when employed in executable code these techniques make the inner workings of the software difficult to analyze and understand. The Binary Analysis Framework will not attempt to detect intentional malformations, obfuscation or

`

anti-analysis techniques and may be unable to parse the desired file or produce incorrect disassembly listings.

5.4.2 File format and architecture support. The Binary Analysis Framework currently supports a very narrow set of architectures and file formats. Namely, the portable executable file format and 32-bit intel architecture. While the framework provides for expansion to include other file formats and architecture, it would require additional system development to implement. This would require background knowledge in Java development, software engineering and binary analysis.

5.4.3 User interface. The user interface provides basic information concerning the analyzed files and search results. This information includes virtual address, mnemonic, operands and function location. The user does not have the ability to easily navigate output, rename virtual addresses, rename operands or provide comments. These features would increase the effectiveness of the framework and are supported by the architectural design, but require implementation.

5.5 Summary

This chapter provided a summary of the work presented. Section 5.1 detailed significant contributions from this work, which include efficiencies in program analysis, project-based user interface and a novel search architecture. Lessons learned during artifact development are highlighted in section 5.2. Future work is discussed in section 5.3 and followed in section 5.4 by current framework limitations. The appendices contain database schema design, framework performance analysis and complete code listings for critical application files.

`

REFERENCES

4. Using FXML to Create a User Interface. (n.d.). Retrieved from http://docs.oracle.com/javafx/2/get_started/fxml_tutorial.htm

Adobe Flash. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Adobe_Flash

Allain, A. (n.d.). Compiling and Linking. Retrieved from http://www.cprogramming.com/compilingandlinking.html

Andriesse, D., Chen, X., van der Veen, V., Slowinska, A., & Bos, H. (2016). *An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries.* Paper presented at the USENIX Security Symposium.

angr. (n.d.). Retrieved from http://angr.io/

Automatic Memory Management. (n.d.). Retrieved from https://msdn.microsoft.com/en-us/library/f144e03t(v=vs.110).aspx

Common Vulnerabilities and Exposures. (n.d.). Retrieved from https://cve.mitre.org/

Computer Performance. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Computer_performance

Cost of data breaches increasing to average of $3.8 million, study says. (2015, May 27). Retrieved from http://www.reuters.com/article/us-cybersecurity-ibm-idUSKBN0OC0ZE20150527

Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., . . . Hinton, H. (1998). *StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks.* Paper presented at the Usenix Security.

`

CWE-415: Double Free. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Computer_performance

Data Breach. (2016). Retrieved from http://www.trendmicro.com/vinfo/us/security/definition/data-breach

Davies, A. (2014). Infiniti's New Steering System Is a Big Step Forward—Unless You Love Cars. Retrieved from http://www.wired.com/2014/06/infiniti-q50-steer-by-wire/

Dowd, M., McDonald, J., & Schuh, J. (2006). *The art of software security assessment: Identifying and preventing software vulnerabilities*: Pearson Education.

Drago, E. (2015). The effect of technology on face-to-face communication. *The Elon Journal of Undergraduate Research in Communications, 6*(1), 13-19.

Eagle, C. (2011). *The IDA pro book: the unofficial guide to the world's most popular disassembler*: No Starch Press.

The Eight Most Common Causes of Data Breaches. (2013, May 22). Retrieved from http://www.darkreading.com/attacks-breaches/the-eight-most-common-causes-of-data-breaches/d/d-id/1139795

The ENIAC vs The Cell Phone. (n.d.). Retrieved from http://www.antiquetech.com/?page_id=1438

Erickson, J. (2008). *Hacking: the art of exploitation*: No Starch Press.

Executable and Linkable Format. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

Executable Space Protection. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Executable_space_protection - Windows

`

File Format. (n.d.). Retrieved from https://en.wikipedia.org/wiki/File_format - Magic_number

Flores, C. (2015, February 18). Most vulnerable operating systems and applications in 2014. Retrieved from http://www.gfi.com/blog/most-vulnerable-operating-systems-and-applications-in-2014/

Francis, K. A., & Ginsberg, W. (2016). The Federal Cybersecurity Workforce: Background and Congressional Oversight Issues for the Departments of Defense and Homeland Security.

Gawlik, R., & Holz, T. (2014). *Towards automated integrity protection of C++ virtual function tables in binary programs.* Paper presented at the Proceedings of the 30th Annual Computer Security Applications Conference.

/GS (Buffer Security Check). (n.d.). Retrieved from https://msdn.microsoft.com/en-us/library/8dbf701c.aspx

Hahn, K., & Register, I. (2014). *Robust Static Analysis of Portable Executable Malware.*

Hamblen, M. (2009, Mar 14). Cell phone, smartphone -- what's the difference? Retrieved from http://www.computerworld.com/article/2531555/mobile-wireless/cell-phone--smartphone----what-s-the-difference-.html

Hardware Control Flow Integrity (CFI) for an IT Ecosystem. (2015). (pp. 19): NSA Information Assurance.

Hevner, A., March, S., Park, J., & Ram, S. (2004). Design Science in Information System Research. *MIS Quarterly, 28*(1), 75-105.

History of Computing Hardware. (n.d.). Retrieved from https://en.wikipedia.org/wiki/History_of_computing_hardware - First_general-purpose_computing_device

History of Laptops. (n.d.). Retrieved from https://en.wikipedia.org/wiki/History_of_laptops

`

Huddleston, T. (2015, September 15, 2015). This graphic shows all the ways your car can be hacked. Retrieved from http://fortune.com/2015/09/15/intel-car-hacking/

IMAGE_OPTIONAL_HEADER structure. (n.d.). Retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/ms680339(v=vs.85).aspx

Intel 64 and IA-32 Architectures Software Developer's Manual. (n.d.). Vol. 2. (pp. 1515). Retrieved from http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

Jamthagen, C., Lantz, P., & Hell, M. (2013). *A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries.* Paper presented at the Anti-malware Testing Research (WATeR), 2013 Workshop on.

Java Virtual Machine. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Java_virtual_machine

Kassner, M. (2015, February 2). Anatomy of the Target Data Breach: MIssed opportunities and lessons learned. Retrieved from http://www.zdnet.com/article/anatomy-of-the-target-data-breach-missed-opportunities-and-lessons-learned/

Krahmer, S. (2005). x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique.

Krebs, B. (2014). The Target Breach, By the Numbers. Retrieved from https://krebsonsecurity.com/2014/05/the-target-breach-by-the-numbers/

Linker (computing). (2016). Retrieved from https://en.wikipedia.org/wiki/Linker_%28computing%29

`

Linn, C., & Debray, S. (2003). *Obfuscation of executable code to improve resistance to static disassembly.* Paper presented at the Proceedings of the 10th ACM conference on Computer and communications security.

Malware report: The economic impact of viruses, spyware, adware, botnets, and other malicious code. (2007). Retrieved from http://www.computereconomics.com/article.cfm?id=1225

Mandiant. (2016). *M-Trends 2016*. Retrieved from FireEye: https://www.fireeye.com/current-threats/annual-threat-report/mtrends.html

Manes, C. (2016, April 6). 2015's MVPs - The most vulnerable players. Retrieved from http://www.gfi.com/blog/2015s-mvps-the-most-vulnerable-players/

Martin, R. C. (2009). *Clean code: a handbook of agile software craftsmanship*: Pearson Education.

Memory Corruption. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Memory_corruption

Merces, F., & Weyrich, J. (2017). pev - The PE file analysis toolkit. Sourceforge. Retrieved from http://pev.sourceforge.net/

Microsoft. (1999). *Microsoft Portable Executable and Common Object File Format Specification* Retrieved from https://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/pecoff.doc

Microsoft. (n.d.). Control Flow Guard. Retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx

Mo, R., Cai, Y., Kazman, R., Xiao, L., & Feng, Q. (2016). *Decoupling level: a new metric for architectural maintenance complexity.* Paper presented at the Proceedings of the 38th International Conference on Software Engineering.

`

Morgan, J. (2014). A Simple Explanation of 'The Internet Of Things'. Retrieved from http://www.forbes.com/sites/jacobmorgan/2014/05/13/simple-explanation-internet-things-that-anyone-can-understand/ - 63635c626828

N-Tier Data Applications Overview. (n.d.). Retrieved from https://msdn.microsoft.com/en-us/library/bb384398.aspx

Namespace. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Namespace

National Vulnerability Database. (2016). Retrieved from https://nvd.nist.gov/

One, A. (1996). Smashing the stack for fun and profit. *Phrack magazine, 7*(49), 14-16.

Operating system market share. (2016). Retrieved from https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10

Operating Systems. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Operating_system

Oracle. (n.d.). Your First Cup. Retrieved from http://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html

Paleari, R., Martignoni, L., Fresi Roglia, G., & Bruschi, D. (2010). *N-version disassembly: differential testing of x86 disassemblers.* Paper presented at the Proceedings of the 19th international symposium on Software testing and analysis.

Popa, M. (2012). Binary Code Disassembly for Reverse Engineering. *Journal of Mobile, Embedded and Distributed Systems, 4*(4), 233-248.

Ray, B., Posnett, D., Filkov, V., & Devanbu, P. (2014). *A large scale study of programming languages and code quality in github.* Paper presented at the Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.

Regalado, D., Harris, S., Harper, A., Eagle, C., Ness, J., Spasojevic, B., . . . Sims, S. (2015). *Gray Hat Hacking the Ethical Hacker's Handbook*: McGraw-Hill Education Group.

`

Reimer, J. (2005, 12/14/2005). Total Share: 30 Years of Personal Computer Market Share Figures. Retrieved from http://arstechnica.com/features/2005/12/total-share/10/

Russinovich, M. E., Solomon, D. A., & Ionescu, A. (2012). *Windows internals*: Pearson Education.

Schaller, R. R. (1997). Moore's law: past, present and future. *IEEE Spectrum, 34*(6), 52-59. doi:10.1109/6.591665

Schmalz, M. S. (n.d.). Organization of Computer Systems: ISA, Machine Language, Number Systems. Retrieved from https://www.cise.ufl.edu/~mssz/CompOrg/CDA-lang.html

Sikorski, M., & Honig, A. (2012). *Practical Malware Analysis: The hands-on guide to dissecting malicious software*: No Starch Press.

Smartphone. (n.d.). Retrieved from https://en.wikipedia.org/wiki/Smartphone

Song, C., Lee, B., Lu, K., Harris, W. R., Kim, T., & Lee, W. (2016). *Enforcing kernel security invariants with data flow integrity.* Paper presented at the Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., . . . Vigna, G. (2016). *Driller: Augmenting fuzzing through selective symbolic execution.* Paper presented at the Proceedings of the Network and Distributed System Security Symposium.

STIC, P. (2017). BARF: A multiplatform open source binary analysis and reverse engineering framework. Github: Programa STIC. Retrieved from https://github.com/programa-stic/barf-project

Technological Development and Dependency. (2011). Retrieved from https://www.fema.gov/pdf/about/programs/oppa/technology_dev_ paper.pdf.

`

Virtual Address Space. (n.d.). Retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/aa366912(v=vs.85).aspx

Wartell, R., Zhou, Y., Hamlen, K. W., Kantarcioglu, M., & Thuraisingham, B. (2011). Differentiating code from data in x86 binaries *Machine Learning and Knowledge Discovery in Databases* (pp. 522-536): Springer.

What is a DLL? (n.d.). Retrieved from https://support.microsoft.com/en-us/kb/815065

Wojdyla, B. (2013, Feb 21, 2012). Retrieved from http://www.popularmechanics.com/cars/how-to/a7386/how-it-works-the-computer-inside-your-car/

Worthen, B. (2010). Rising Computer Prices Buck The Trend. Retrieved from http://www.wsj.com/articles/SB10001424052748704681804576017883787191962

You, I., & Yim, K. (2010). *Malware obfuscation techniques: A brief survey*. Paper presented at the Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on.

Zakas, N. C. (2013). The evolution of web development for mobile devices. *Queue, 11*(2), 30.

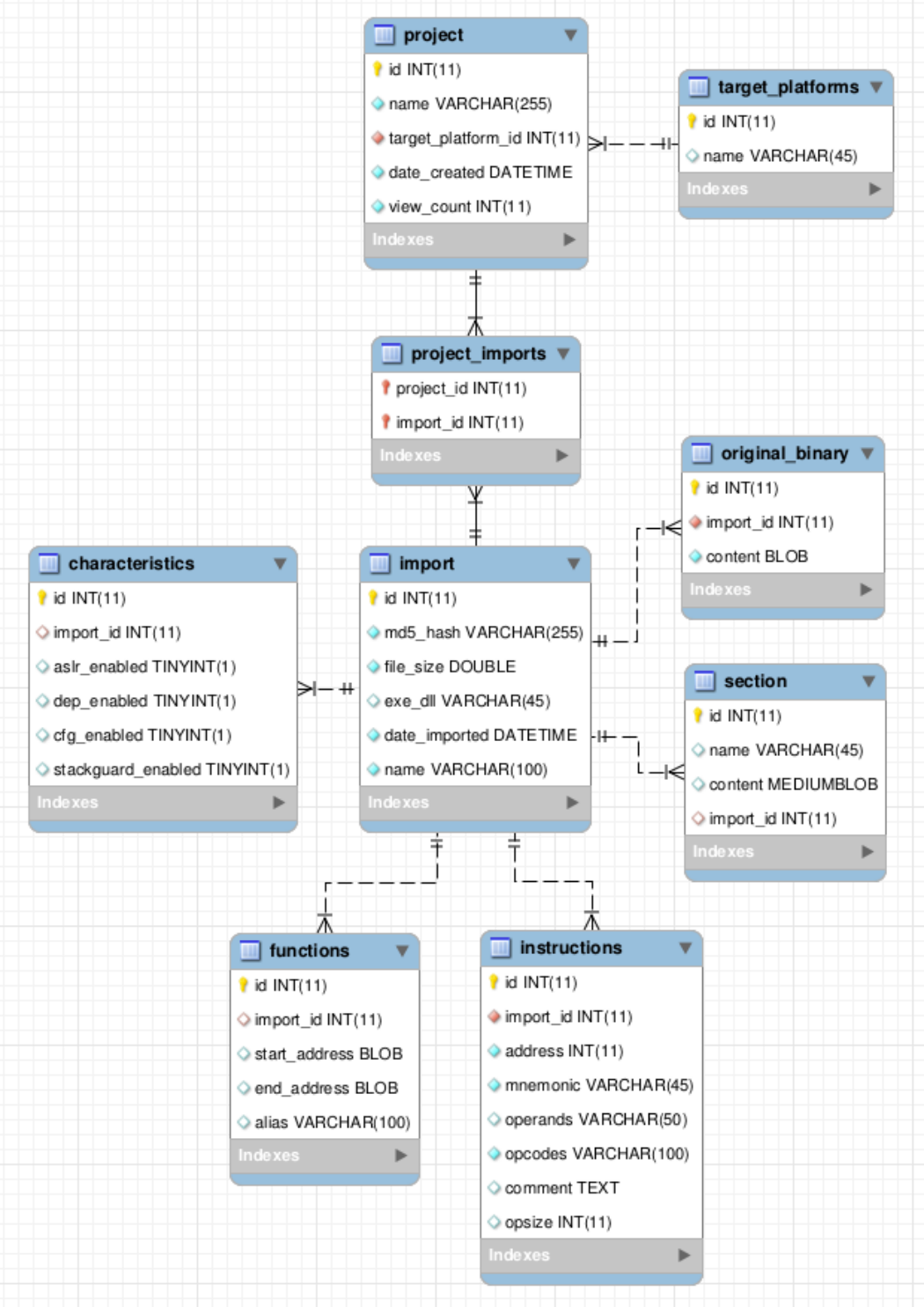
Zwanger, V., Gerhards-Padilla, E., & Meier, M. (2014). *Codescanner: Detecting (Hidden) x86/x64 code in arbitrary files.* Paper presented at the Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on.

`

APPENDICES

`

APPENDIX A: DATABASE ARCHITECTURE



`

Table 2. Description of Application Database Classes

Class	Member/Member Function	Description
DBConnection	Constructor	Provides instantiation of connection object used by later database access objects.
Project	Constructors	Provide population of object members during object instantiation.
	ID	Primary key of the data, used by database for data normalization.
	ProjectName	String representing the name of the project, user defined.
	TargetPlatform	The desired platform of the project.
	DateCreated	The date and time that the project was created.
	ViewCount	A numeric value incremented each time the project is loaded.
	Save()	Member function that provides conditional logic to determine when to insert new data or update an existing record based on presence of the primary key member ID.
	GetAllProjects()	Static method that returns a list of all Project objects.
Import	Constructors	Provide population of object members during object instantiation. Multiple constructors are provided.
	ID	Primary key of the data, used by database for data normalization.
	FileSize	Size in bytes of imported file.
	DateCreated	The date and time that the project was created.
	Name	Name of imported file.
	functions	List of *ShortenedFunction* objects that represent all functions identified during disassembly.
	instructions	List of *ShortenedInsn* objects that represent all instructions identified during disassembly.

`

	Save()	Member function that provides conditional logic to determine when to insert new data or update an existing record based on presence of the primary key member ID.
	SaveRawBytes()	Saves the original binary content of the file by inserting into the *original_binary* table.
	SaveFunctions()	Iterates the *functions* member and inserts data into the *functions* table.
	SaveInstructions()	Iterates the *instructions* member and inserts data into the *instructions* table.
	buildMnemonicExpression()	Static method that builds an SQL statement using regular expression syntax based on input provided in the search form.
	searchByMnemonicExpression()	Static method that utilizes *buildMnemonicExpression* function to construct an SQL statement to search *instructions* table for desired assembly sequence. This searches a single instruction at a time.
	searchByMnemonicSequence()	Static method that utilizes *buildMnemonicExpression* function to construct an SQL statement to search *instructions* table for desired assembly sequence. This searches multiple instructions at a time.
	getFunctionInstructions()	Member function that returns all the instructions associated with a specific function. Function information is determend by using an instance of the *ShortenedFunction* object passed as a single argument.
	getAllImports()	Static method that returns all imports related to a specific project. Expects one argument which is the desired project ID.
	searchRopGadgets()	Static method that returns a list collection of string objects that represent discovered return-oriented programming gadgets.
Characteristics	Constructors	Provide population of object members during object instantiation. Multiple constructors are provided.
	Save()	Member function that provides conditional logic to determine when to insert new data or update an existing record based on presence of the primary key member ID.
Section	Constructors	Provide population of object members during object instantiation. Multiple constructors are provided.
	Save()	Member function that provides conditional logic to determine when to insert new data or update an existing record based on presence of the primary key member ID.

`

Original_Binary	Constructors	Provide population of object members during object instantiation. Multiple constructors are provided.
	Save()	Member function that provides conditional logic to determine when to insert new data or update an existing record based on presence of the primary key member ID.

`

Table 3. Performance Analysis

Library Name	File Size	Load Time (seconds)	ASLR	DEP	CFG	Stack Guard
advapi32.dll	640.5 KB	225	x	x		
crypt32.dll	1.2 MB	556	x	x		
cryptbase.dll	36.9 KB	12	x	x		
kernel32.dll	857.6 KB	726	x	x		
ntdll.dll	1.3 MB	825	x	x		
winhttp.dll	351.2 KB	107	x	x		
wininet.dll	981.0 KB	601	x	x		

`

APPENDIX B: BINARY ANALYSIS FRAMEWORK CODE

Controller.java

```
1.  package gui;
2.
3.  import java.io.BufferedReader;
4.  import java.io.File;
5.  import java.io.FileNotFoundException;
6.  import java.io.FileReader;
7.  import java.io.IOException;
8.  import java.util.ArrayList;
9.  import java.util.List;
10. import java.util.ListIterator;
11. import java.util.Map;
12. import java.util.TreeMap;
13.
14. import org.apache.commons.io.FilenameUtils;
15.
16. import com.thoughtworks.xstream.XStream;
17. import com.thoughtworks.xstream.io.xml.DomDriver;
18.
19. import application.database.Characteristics;
20. import application.database.Import;
21. import application.database.Project;
22. import application.database.section;
23. import edu.millermj.disassembler.Code;
24. import edu.millermj.disassembler.Function;
25. import edu.millermj.disassembler.Ins;
26. import gui.service.DisassembleService;
27. import gui.service.FileChooseService;
28. import javafx.collections.FXCollections;
29. import javafx.collections.ObservableList;
30. import javafx.event.ActionEvent;
31. import javafx.event.EventHandler;
32. import javafx.fxml.FXMLLoader;
33. import javafx.geometry.Insets;
34. import javafx.geometry.Orientation;
35. import javafx.scene.Parent;
36. import javafx.scene.Scene;
37. import javafx.scene.control.Label;
38. import javafx.scene.control.ListCell;
39. import javafx.scene.control.ListView;
40. import javafx.scene.control.Menu;
41. import javafx.scene.control.MenuItem;
42. import javafx.scene.control.SplitPane;
43. import javafx.scene.control.Tab;
44. import javafx.scene.control.TabPane;
45. import javafx.scene.control.TableColumn;
46. import javafx.scene.control.TableView;
47. import javafx.scene.control.TextArea;
48. import javafx.scene.control.TextField;
49. import javafx.scene.control.cell.PropertyValueFactory;
50. import javafx.scene.layout.Background;
51. import javafx.scene.layout.BackgroundFill;
52. import javafx.scene.layout.CornerRadii;
53. import javafx.scene.layout.HBox;
```

`

```
54. import javafx.scene.paint.Color;
55. import javafx.stage.Modality;
56. import javafx.stage.Popup;
57. import javafx.stage.Stage;
58. import javafx.stage.StageStyle;
59. import javafx.util.Callback;
60.
61. public class Controller  {
62.
63.     public MenuItem loadItem;
64.     public Menu searchMenu;
65.     public MenuItem importBinary;
66.     public Menu infoMenu;
67.     //public TableView tableView;
68.     public TableColumn addressCol;
69.     public TableColumn opCodeCol;
70.     public TableColumn mnemonicCol;
71.     public TableColumn operandCol;
72.     public ListView searchListView;
73.
74.     public TabPane tabPane;
75.
76.     public TextArea outputLog;
77.     public Project loadedProject;
78.
79.     InteractiveInstruction interactiveInstruction;
80.
81.     private String filename;
82.     private GUI gui;
83.
84.     public enum MESSAGE_TYPE {
85.         INFO, WARNING, ERROR
86.     }
87.
88.     public void setGui(GUI gui) {
89.         this.gui = gui;
90.
91.         /*listView.setOnMouseClicked(new EventHandler<MouseEvent>() {
92.             @Override
93.             public void handle(MouseEvent event) {
94.                 if(event.getButton() == MouseButton.SECONDARY) {
95.                     showFunctionMenu((ShortenedFunction) listView.getSelectionModel().g
    etSelectedItem());
96.                 }
97.             }
98.         });*/
99.     }
100.
101.     public void openNew() {
102.
103.         logOutput("Importing new binary file...", MESSAGE_TYPE.INFO);
104.
105.         long startTime = System.currentTimeMillis();
106.
107.         File file = FileChooseService.chooseFile(gui.getStage(), "new");
108.         if(file == null) {
109.             return;
110.         }
111.
112.         Project p = gui.getLoadedProject();
```

`

```
113.
114.        //Create new import
115.        Import binary = new Import();
116.        binary.setFileSize(file.length());
117.        binary.setName(file.getName());
118.        binary.setMD5Hash("hash");
119.        binary.setProjectID(p.getID());
120.
121.        logOutput("New binary imported", MESSAGE_TYPE.INFO);
122.
123.        this.filename = FilenameUtils.removeExtension(file.getName());
124.
125.        //Store the original bytes
126.        binary.SaveRawBytes(file);
127.
128.        logOutput("Original bytes stored", MESSAGE_TYPE.INFO);
129.
130.        Code c = DisassembleService.getCode(file);
131.
132.        Characteristics chars = new Characteristics();
133.
134.        chars.setASLREnabled(c.getASLREnabled());
135.        chars.setDEPEnabled(c.getDEPEnabled());
136.        chars.setCFGEnabled(c.getCFGEnabled());
137.        chars.setStackGaurdEnabled(c.getStackGuardEnabled());
138.
139.        binary.setCharacteristics(chars);
140.
141.        binary.Save();
142.
143.        section codeSection = new section();
144.
145.        codeSection.content = c.getCodeSectionBytesAsString();
146.        codeSection.name = c.getCodeSegmentName();
147.        codeSection.import_id = binary.getID();
148.        codeSection.save();
149.
150.        TreeMap<Integer, ShortenedFunction> shortFunctionMap = new TreeMap<Integer, Sho
    rtenedFunction>();
151.        TreeMap<Integer, ShortenedIns> shortInsMap = new TreeMap<Integer, ShortenedIns>
    ();
152.        List<ShortenedFunction> functions = new ArrayList<ShortenedFunction>();
153.        List<ShortenedIns> instructions = new ArrayList<ShortenedIns>();
154.
155.        for(Map.Entry<Integer, Function> entry : c.getFunctionMap().entrySet()) {
156.            ShortenedFunction shortenedFunction = new ShortenedFunction(entry.getValue(
    ));
157.            shortFunctionMap.put(entry.getKey(), shortenedFunction);
158.
159.            functions.add(shortenedFunction);
160.        }
161.
162.        for(Map.Entry<Integer, Ins> entry : c.getCodeMap().entrySet()) {
163.            ShortenedIns shortenedIns = new ShortenedIns(entry.getValue());
164.            shortInsMap.put(entry.getKey(), shortenedIns);
165.
166.            instructions.add(shortenedIns);
167.        }
168.
169.        binary.setFunctions(functions);
```

`

```
170.         binary.SaveFunctions();
171.
172.         binary.setInstructions(instructions);
173.         binary.SaveInstructions();
174.
175.         p.imports  = Import.GetAllImports(p.getID());
176.
177.         gui.setLoadedProject(p);
178.
179.         fillDisplay();
180.
181.         logOutput("Binary Loaded, total time: " + (System.currentTimeMillis() - startTi
    me)/1000 + "s", MESSAGE_TYPE.INFO);
182.     }
183.
184.     public void loadFile(){
185.
186.         long startTime = System.currentTimeMillis();
187.         File file = FileChooseService.chooseFile(gui.getStage(), "load");
188.
189.         if(file == null) {
190.             return;
191.         }
192.
193.         this.filename = FilenameUtils.removeExtension(file.getName());
194.
195.         XStream xStream = new XStream(new DomDriver());
196.         StringBuilder stringBuilder = new StringBuilder();
197.         try {
198.             BufferedReader reader = new BufferedReader( new FileReader (file.getPath())
    );
199.             String line = null;
200.
201.             while((line = reader.readLine()) != null) {
202.                 stringBuilder.append(line);
203.             }
204.         } catch (FileNotFoundException e) {
205.             e.printStackTrace();
206.         } catch (IOException e) {
207.             e.printStackTrace();
208.         }
209.
210.
211.         interactiveInstruction = (InteractiveInstruction) xStream.fromXML(stringBuilder
    .toString());
212.
213.         fillDisplay();
214.
215.         //System.out.println("loadFile Done: " +(endTime - startTime)/1000);
216.         outputLog.appendText("Binary Loaded, total time: " + (System.currentTimeMillis(
    ) - startTime)/1000);
217.
218.     }
219.
220.     public void fillDisplay() {
221.
222.         List<Import>imports = gui.getLoadedProject().imports;
223.         logOutput("Number of imports: " + imports.size(), MESSAGE_TYPE.INFO);
224.
225.         if (imports.size() > 0) {
```

`

```
226.             searchMenu.setDisable(false);
227.         }
228.
229.         //Clear Tab Pane
230.         tabPane.getTabs().clear();
231.
232.         for(ListIterator<Import> li = imports.listIterator(); li.hasNext();){
233.             Import i = li.next();
234.
235.             List<ShortenedFunction> funcs = i.getFunctions();
236.             List<ShortenedIns> inst = i.getInstructions();
237.
238.             Tab newTab = new Tab(i.getName());
239.
240.             SplitPane mainPane = new SplitPane();
241.             mainPane.setOrientation(Orientation.HORIZONTAL);
242.
243.
244.             SplitPane leftPane = new SplitPane();
245.             leftPane.setMinWidth(100);
246.             leftPane.setMaxWidth(100);
247.
248.             leftPane.setOrientation(Orientation.VERTICAL);
249.
250.             Label l = new Label();
251.
252.             l.setText("Functions");
253.             l.prefHeight(100);
254.
255.             leftPane.getItems().add(l);
256.
257.             ObservableList<ShortenedFunction> obsFunctions = FXCollections.observableAr
    rayList(funcs);
258.
259.             ListView tabListView = new ListView();
260.             tabListView.setPrefWidth(100);
261.             tabListView.setPrefHeight(70);
262.             tabListView.setItems(obsFunctions);
263.             tabListView.setCellFactory(new Callback<ListView, ListCell<ShortenedFunctio
    n>>() {
264.                 @Override
265.                 public ListCell<ShortenedFunction> call(ListView param) {
266.                     return new ListCell<ShortenedFunction>() {
267.                         @Override
268.                         public void updateItem(ShortenedFunction fn, boolean empty) {

269.                             super.updateItem(fn, empty);
270.                             if (!isEmpty()) {
271.                                 this.setText(fn.getAlias());
272.                             }
273.                         }
274.                     };
275.                 }
276.             });
277.
278.             leftPane.getItems().add(tabListView);
279.
280.             mainPane.getItems().add(leftPane);
281.
282.             //RIGHT Pane - TableView
```

`

```
283.             TableView newTabTable = new TableView();
284.             addressCol = new TableColumn();
285.             addressCol.setText("Address");
286.             opCodeCol = new TableColumn();
287.             opCodeCol.setText("OP Codes");
288.             mnemonicCol = new TableColumn();
289.             mnemonicCol.setText("Mnemonic");
290.             operandCol = new TableColumn();
291.             operandCol.setText("Operands");
292.
293.             ObservableList<ShortenedIns> obsInstructions = FXCollections.observableArra
    yList(inst);
294.
295.             addressCol.setCellValueFactory(new PropertyValueFactory<Ins, String>("addre
    ssString"));
296.             opCodeCol.setCellValueFactory(new PropertyValueFactory<Ins, String>("opcode
    S"));
297.             mnemonicCol.setCellValueFactory(new PropertyValueFactory<Ins, String>("mnem
    onic"));
298.             operandCol.setCellValueFactory(new PropertyValueFactory<Ins, String>("opera
    nds"));
299.
300.             newTabTable.setItems(obsInstructions);
301.             newTabTable.getColumns().addAll(addressCol, opCodeCol, mnemonicCol, operand
    Col);
302.             newTabTable.refresh();
303.
304.             mainPane.getItems().add(newTabTable);
305.
306.             newTab.setContent(mainPane);
307.
308.             newTab.setId(Integer.toString(i.getID()));
309.
310.             tabPane.getTabs().add(newTab);
311.         }
312.     }
313.
314.     public void saveFile() {
315.         FileChooseService.saveFile(gui.getStage(), filename, interactiveInstruction);

316.     }
317.
318.     public void loadProject() {
319.
320.         Parent root;
321.         FXMLLoader loader;
322.
323.         try {
324.
325.             Stage primary_stage = gui.getStage();
326.
327.             Stage project_dialog = new Stage(StageStyle.UTILITY);
328.
329.             project_dialog.initModality(Modality.WINDOW_MODAL);
330.             project_dialog.initOwner(primary_stage);
331.
332.             loader = new FXMLLoader(getClass().getResource("ProjectLoaderView.fxml"));

333.             root = loader.load();
334.
```

`

```
335.            ProjectLoaderController projectController = loader.getController();
336.
337.            projectController.setGui(gui);
338.
339.            projectController.update_project_list();
340.
341.            Scene scene = new Scene(root);
342.            project_dialog.setScene(scene);
343.
344.            project_dialog.setTitle("Choose A Project");
345.            project_dialog.showAndWait();
346.
347.            fillDisplay();
348.            /*project_dialog.setOnCloseRequest(new EventHandler<WindowEvent>() {
349.                public void handle(WindowEvent we) {
350.                    logOutput("Project Dialog is closing...", MESSAGE_TYPE.INFO);
351.
352.                }
353.            });*/
354.
355.            importBinary.setDisable(false);
356.            infoMenu.setDisable(false);
357.
358.        } catch(Exception ex) {
359.            System.out.print("ERROR: "+ex.toString());
360.        }
361.    }
362.
363.    public void showFunctionMenu(ShortenedFunction fn) {
364.        /*final ContextMenu contextMenu = new ContextMenu();
365.        final MenuItem rename = new MenuItem("Rename");
366.
367.        rename.setOnAction(new EventHandler<ActionEvent>() {
368.            @Override
369.            public void handle(ActionEvent event) {
370.                showRenamePopup(fn);
371.            }
372.        });
373.
374.        contextMenu.getItems().add(rename);
375.        listView.setContextMenu(contextMenu);*/
376.    }
377.
378.    public void showRenamePopup(ShortenedFunction fn) {
379.        final Popup popup = new Popup();
380.        final HBox renameContainer = new HBox();
381.        renameContainer.setBackground(new Background(new BackgroundFill(Color.WHEAT, Co
    rnerRadii.EMPTY, Insets.EMPTY)));
382.        final Label label = new Label("Rename " + fn.getAlias() + " to:");
383.        label.setPadding(new Insets(2, 5, 2, 5));
384.        final TextField textField = new TextField();
385.
386.        textField.setOnAction(new EventHandler<ActionEvent>() {
387.            @Override
388.            public void handle(ActionEvent event) {
389.                for(Map.Entry<Integer, ShortenedIns> ins: interactiveInstruction.getIns
    Map().entrySet()) {
390.                    if(ins.getValue().getOperands().equals(fn.getAlias())) {
391.                        ins.getValue().setOperands(textField.getText());
392.                    }
```

`

```
393.
394.                 }
395.                 interactiveInstruction.getFnMap().get(fn.getStartAddress()).setAlias(te
    xtField.getText());
396.
397.                 fillDisplay();
398.                 popup.hide();
399.             }
400.         });
401.         renameContainer.getChildren().addAll(label, textField);
402.
403.         popup.getContent().add(renameContainer);
404.         popup.show(gui.getStage());
405.     }
406.
407.     public void searchROPGadgets() {
408.
409.         Parent root;
410.         FXMLLoader loader;
411.
412.         try {
413.
414.             Tab selectedTab = tabPane.getSelectionModel().getSelectedItem();
415.
416.             gui.setSelectedImportID(Integer.parseInt(selectedTab.getId()));
417.
418.             Stage primary_stage = gui.getStage();
419.
420.             Stage search_window  = new Stage(StageStyle.UTILITY);
421.
422.             search_window.initModality(Modality.WINDOW_MODAL);
423.             search_window.initOwner(primary_stage);
424.
425.             loader = new FXMLLoader(getClass().getResource("ROPSearch.fxml"));
426.             root = loader.load();
427.
428.             ROPSearchController ropController = loader.getController();
429.
430.             ropController.setGui(gui);
431.
432.             Scene scene = new Scene(root);
433.             search_window.setScene(scene);
434.
435.             search_window.setTitle("Search For ROP Gadgets");
436.             search_window.showAndWait();
437.
438.         }catch(Exception ex) {
439.             System.out.print("ERROR: " + ex.toString());
440.         }
441.     }
442.
443.     public void searchFunctions() {
444.
445.         Parent root;
446.         FXMLLoader loader;
447.
448.         try {
449.
450.             Tab selectedTab = tabPane.getSelectionModel().getSelectedItem();
451.
```

`

```
452.             gui.setSelectedImportID(Integer.parseInt(selectedTab.getId()));
453.
454.             Stage primary_stage = gui.getStage();
455.
456.             Stage search_window  = new Stage(StageStyle.UTILITY);
457.
458.             search_window.initModality(Modality.WINDOW_MODAL);
459.             search_window.initOwner(primary_stage);
460.
461.             loader = new FXMLLoader(getClass().getResource("FunctionSearch.fxml"));
462.             root = loader.load();
463.
464.             FunctionSearchController searchController = loader.getController();
465.
466.             searchController.setGui(gui);
467.
468.             Scene scene = new Scene(root);
469.             search_window.setScene(scene);
470.
471.             search_window.setTitle("Search For Functions By Mnemonic Across All Imports
    ");
472.             search_window.showAndWait();
473.
474.         }catch(Exception ex) {
475.             System.out.print("ERROR: " + ex.toString());
476.         }
477.     }
478.
479.     public void displaySecurityInfo() {
480.
481.         Parent root;
482.         FXMLLoader loader;
483.
484.         try {
485.
486.             Tab selectedTab = tabPane.getSelectionModel().getSelectedItem();
487.
488.             gui.setSelectedImportID(Integer.parseInt(selectedTab.getId()));
489.
490.             Stage primary_stage = gui.getStage();
491.
492.             Stage search_window  = new Stage(StageStyle.UTILITY);
493.
494.             search_window.initModality(Modality.WINDOW_MODAL);
495.             search_window.initOwner(primary_stage);
496.
497.             loader = new FXMLLoader(getClass().getResource("SecurityInfo.fxml"));
498.             root = loader.load();
499.
500.             SecurityInfoController searchController = loader.getController();
501.
502.             searchController.setGui(gui);
503.             searchController.SetupStage();
504.
505.             Scene scene = new Scene(root);
506.             search_window.setScene(scene);
507.
508.             search_window.setTitle("Compiler Options");
509.             search_window.showAndWait();
510.
```

`

```
511.        }catch(Exception ex) {
512.            System.out.print("ERROR: " + ex.toString());
513.        }
514.    }
515.
516.    public void searchByMnemonicExpression() {
517.
518.        Parent root;
519.        FXMLLoader loader;
520.
521.        try {
522.
523.            Tab selectedTab = tabPane.getSelectionModel().getSelectedItem();
524.
525.            gui.setSelectedImportID(Integer.parseInt(selectedTab.getId()));
526.
527.            Stage primary_stage = gui.getStage();
528.
529.            Stage search_window  = new Stage(StageStyle.UTILITY);
530.
531.            search_window.initModality(Modality.WINDOW_MODAL);
532.            search_window.initOwner(primary_stage);
533.
534.            loader = new FXMLLoader(getClass().getResource("MnemonicSearch.fxml"));
535.            root = loader.load();
536.
537.            SearchController searchController = loader.getController();
538.
539.            searchController.setGui(gui);
540.
541.            Scene scene = new Scene(root);
542.            search_window.setScene(scene);
543.
544.            search_window.setTitle("Search By Expression");
545.            search_window.showAndWait();
546.
547.        }catch(Exception ex) {
548.            System.out.print("ERROR: " + ex.toString());
549.        }
550.    }
551.
552.    public void searchByMnemonicSequence() {
553.
554.        Parent root;
555.        FXMLLoader loader;
556.
557.        try {
558.
559.            Tab selectedTab = tabPane.getSelectionModel().getSelectedItem();
560.
561.            gui.setSelectedImportID(Integer.parseInt(selectedTab.getId()));
562.
563.            Stage primary_stage = gui.getStage();
564.
565.            Stage search_window  = new Stage(StageStyle.UTILITY);
566.
567.            search_window.initModality(Modality.WINDOW_MODAL);
568.            search_window.initOwner(primary_stage);
569.
```

`

```
570.             loader = new FXMLLoader(getClass().getResource("MnemonicSequenceSearch.fxml
    "));
571.             root = loader.load();
572.
573.             SearchSequenceController searchController = loader.getController();
574.
575.             searchController.setGui(gui);
576.
577.             Scene scene = new Scene(root);
578.             search_window.setScene(scene);
579.
580.             search_window.setTitle("Search By Sequence");
581.             search_window.showAndWait();
582.
583.         }catch(Exception ex) {
584.             System.out.print("ERROR: " + ex.toString());
585.         }
586.
587.     }
588.
589.
590.     protected void logOutput(String message, MESSAGE_TYPE mType) {
591.
592.         String msgPrefix = "";
593.
594.         switch(mType) {
595.             case ERROR:
596.                 msgPrefix = "[!!] ";
597.                 break;
598.
599.             case WARNING:
600.                 msgPrefix = "[!] ";
601.                 break;
602.
603.             case INFO:
604.                 msgPrefix = "[*] ";
605.                 break;
606.         }
607.
608.         outputLog.appendText(msgPrefix + message + "\n");
609.     }
610. }
```

DisassembleService.java

```
1.  package gui.service;
2.
3.  import java.io.File;
4.
5.  import edu.millermj.disassembler.Code;
6.  import edu.millermj.disassembler.DisassembleInterface;
7.  import edu.millermj.disassembler.DisassemblerFactory;
8.  import edu.millermj.disassembler.LoaderFactory;
9.  import edu.millermj.disassembler.LoaderInterface;
10.
11. public class DisassembleService {
```

`

```
12.
13.     public static Code getCode(File file) {
14.
15.         LoaderFactory factory = new LoaderFactory();
16.         LoaderInterface loader = factory.getLoader(file.getAbsolutePath());
17.         DisassemblerFactory disFactory = new DisassemblerFactory();
18.         DisassembleInterface d = disFactory.getDisassembler(loader);
19.         Code c = d.disassemble(loader);
20.
21.         c.setCodeSectionBytes(loader.getCode());
22.
23.         c.setASLREnabled(loader.isASLREnabled());
24.         c.setDEPEnabled(loader.isDEPEnabled());
25.         c.setCFGEnabled(loader.isCFGEnabled());
26.         c.setStackGaurdEnabled(loader.isStackGuardEnabled());
27.
28.         c.setCodeSegmentName(loader.getCodeSegmentName());
29.
30.         d.generateFunctions(c);
31.
32.         return c;
33.     }
34. }
```

Code.java

```
1.  package edu.millermj.disassembler;
2.
3.  import java.util.TreeMap;
4.
5.  public class Code
6.      {
7.          TreeMap<Integer, Ins>             codeMap;
8.
9.          private TreeMap<Integer, Function>    functionMap;
10.         private byte[] codeSectionBytes;
11.         private short characteristics;
12.         private String codeSegmentName;
13.
14.         private Boolean depEnabled;
15.         private Boolean aslrEnabled;
16.         private Boolean cfgEnabled;
17.         private Boolean stackGuardEnabled;
18.
19.         public Code()
20.             {
21.                 codeMap = new TreeMap<Integer, Ins>();
22.                 setFunctionMap(new TreeMap<Integer, Function>());
23.             }
24.
25.         public TreeMap<Integer, Ins> getCodeMap()
26.             {
27.                 return codeMap;
28.             }
29.
30.         public void addFunction(int address, Function f)
```

`

```
            {
                getFunctionMap().put(address, f);
            }

        public TreeMap<Integer, Function> getFunctionMap()
            {
                return functionMap;
            }

        public void setFunctionMap(TreeMap<Integer, Function> functionMap)
            {
                this.functionMap = functionMap;
            }

        public void setCodeSectionBytes(byte[] codeSectionBytes) {
            this.codeSectionBytes = codeSectionBytes;
        }

        public byte[] getCodeSectionBytes() {
            return this.codeSectionBytes;
        }

        public String getCodeSectionBytesAsString() {

            String ret = "";
            for (int i = 0; i < codeSectionBytes.length; i++)
                ret += String.format("%02x ", codeSectionBytes[i]);
            return ret;
        }

        public short getCharacteristics() {
            return characteristics;
        }

        public void setCharacteristics(short character) {
            this.characteristics = character;
        }

        public void setCodeSegmentName(String name){
            codeSegmentName = name;
        }

        public String getCodeSegmentName() {
            return this.codeSegmentName;
        }

        public Boolean getDEPEnabled() {
            return this.depEnabled;
        }

        public void setDEPEnabled(Boolean value) {
            this.depEnabled = value;
        }

        public Boolean getASLREnabled() {
            return this.aslrEnabled;
        }

        public void setASLREnabled(Boolean value) {
            this.aslrEnabled = value;
```

`

```
91.         }
92.
93.         public Boolean getCFGEnabled() {
94.             return this.cfgEnabled;
95.         }
96.
97.         public void setCFGEnabled(Boolean value) {
98.             this.cfgEnabled = value;
99.         }
100.
101.         public Boolean getStackGuardEnabled() {
102.             return this.stackGuardEnabled;
103.         }
104.
105.         public void setStackGaurdEnabled(Boolean value) {
106.             this.stackGuardEnabled = value;
107.         }
108.     }
```

LoaderInterface.java

```
1.  package edu.millermj.disassembler;
2.
3.  import java.nio.file.Files;
4.  import java.nio.file.Path;
5.  import java.nio.file.Paths;
6.
7.  import application.jpe.ImageDataDirectory;
8.  import application.jpe.ImageSectionHeader;
9.
10. public abstract class LoaderInterface
11.     {
12.         byte[] bytes;
13.
14.         public abstract byte[] getCode();
15.
16.         public abstract CODE_TYPE getType();
17.
18.         public static byte[] loadBytes(String path2)
19.             {
20.                 byte[] bytes = null;
21.                 Path path = Paths.get(path2);
22.                 Long startTime = System.currentTimeMillis();
23.                 try
24.                     {
25.                         bytes = Files.readAllBytes(path);
26.                         Long end = System.currentTimeMillis();
27.                         System.out.format("Load time %d File\n", end - startTime);
28.                     }
29.                 catch (Exception e)
30.                     {
31.                         e.printStackTrace();
32.                     }
33.                 return bytes;
34.             }
35.
```

`

```
36.         public void setBytes(byte[] bytes)
37.             {
38.                 this.bytes = bytes;
39.             }
40.
41.         public abstract int getBaseAddress();
42.
43.         public abstract int getStartAddress();
44.
45.         public abstract String getCodeSegmentName();
46.
47.         public abstract Boolean isDEPEnabled();
48.
49.         public abstract Boolean isASLREnabled();
50.
51.         public abstract Boolean isCFGEnabled();
52.
53.         public abstract Boolean isStackGuardEnabled();
54.     }
```

LoaderFactory.java

```
1.  package edu.millermj.disassembler;
2.
3.  public class LoaderFactory
4.      {
5.          public LoaderFactory()
6.              {
7.
8.              }
9.
10.         public LoaderInterface getLoader(String path)
11.         {
12.             LoaderInterface result = null;
13.             byte[] originalBytes = LoaderInterface.loadBytes(path);
14.             if (originalBytes[0] == 'M' && originalBytes[1] == 'Z') // Windows
15.                 {
16.                     result = new PE();
17.                 }
18.             else if (originalBytes[0] == 0x7F) // elf
19.                 {
20.                     result = new ELF();
21.                 }
22.             result.setBytes(originalBytes);
23.             return result;
24.         }
25.     }
```

DisassembleFactory.java

```
1.  package edu.millermj.disassembler;
2.
3.  import capstone.Capstone;
4.
```

`

```
5.  public class DisassemblerFactory
6.      {
7.          public DisassemblerFactory()
8.              {
9.
10.             }
11.
12.         public DisassembleInterface getDisassembler(LoaderInterface l)
13.             {
14.                 DisassembleInterface disass = null;
15.                 int arch = -1;
16.                 int mode = -1;
17.                 if (l.getType() == CODE_TYPE.ARM)
18.                     {
19.                         disass = new ARM();
20.                         arch = Capstone.CS_ARCH_ARM;
21.                         mode = Capstone.CS_MODE_ARM;
22.                     }
23.                 else if (l.getType() == CODE_TYPE.x86)
24.                     {
25.                         disass = new x86();
26.                         arch = Capstone.CS_ARCH_X86;
27.                         mode = Capstone.CS_MODE_32;
28.                     }
29.
30.                 Capstone cs = new Capstone(arch, mode);
31.                 cs.setDetail(Capstone.CS_OPT_ON);
32.                 // cs.setSyntax();
33.                 disass.setCapstone(cs);
34.                 return disass;
35.             }
36.     }
```

PE.java

```
1.  package edu.millermj.disassembler;
2.
3.  import application.jpe.ImageDOSHeader;
4.  import application.jpe.ImageDataDirectory;
5.  import application.jpe.ImageFileHeader;
6.  import application.jpe.ImageNTHeaders;
7.  import application.jpe.ImageSectionHeader;
8.  import application.jpe.LittleEndian;
9.
10. public class PE extends LoaderInterface
11.     {
12.         private ImageDOSHeader  header;
13.
14.         private ImageNTHeaders  fileHeader;
15.         int                     codeOffset  = 0;
16.
17.         byte[]                  code;
18.
19.         private short characteristics = 0;
20.         private String codeSegmentName = "";
21.
22.         int IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE    = 0x0040;
```

`

```
23.         int IMAGE_DLLCHARACTERISTICS_NX_COMPAT      = 0x0100;
24.         int IMAGE_DLLCHARACTERISTICS_NO_SEH         = 0x0400;
25.         int IMAGE_DLLCHARACTERISTICS_GUARD_CF       = 0x4000;
26.
27.         public PE()
28.             {
29.
30.             }
31.
32.         void load()
33.             {
34.
35.                 header = new ImageDOSHeader(bytes);
36.                 header.debugPrint(System.out);
37.                 Integer elf_anew = header.getELFHeaderRVA();
38.                 code = null;
39.                 try
40.                     {
41.                         fileHeader = new ImageNTHeaders(bytes, elf_anew);
42.
43.                         this.characteristics = fileHeader.getImageOptionalHeader().getD
    llCharacteristics();
44.
45.                         ImageSectionHeader codeSegement = fileHeader.getCodeSegement();

46.                         codeOffset = codeSegement.getVirtualAddress();
47.                         codeSegmentName = codeSegement.name;
48.
49.                         code = new byte[codeSegement.sizeOfRawData];
50.                         for (int start = 0, i = codeSegement.pointerToRawData; start <

51.                                 codeSegement.sizeOfRawData; i++, start++)
52.                             {
53.                                 code[start] = bytes[i];
54.                             }
55.                     }
56.                 catch (Exception e)
57.                     {
58.                         e.printStackTrace();
59.                     }
60.             }
61.
62.         @Override
63.         public Boolean isDEPEnabled() {
64.             if((characteristics & IMAGE_DLLCHARACTERISTICS_NX_COMPAT) != 0)
65.                 return true;
66.
67.             return false;
68.         }
69.
70.         @Override
71.         public Boolean isASLREnabled() {
72.
73.             if((characteristics & IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE) != 0)
74.                 return true;
75.
76.             return false;
77.         }
78.
79.         @Override
```

`

```
80.         public Boolean isCFGEnabled() {
81.             if((characteristics & IMAGE_DLLCHARACTERISTICS_GUARD_CF) != 0)
82.                 return true;
83.
84.             return false;
85.         }
86.
87.         @Override
88.         public Boolean isStackGuardEnabled() {
89.
90.             ImageDataDirectory loadConfig = fileHeader.getImageOptionalHeader().getData
    Directory()[10];
91.             ImageSectionHeader rdataSegmentHeader = fileHeader.getSegment(".rdata");
92.
93.             if (rdataSegmentHeader != null) {
94.
95.                 int dataOffset = (int) (loadConfig.getAddress() - rdataSegmentHeader.ge
    tVirtualAddress());
96.
97.                 int foa = (int) rdataSegmentHeader.getRawDataPosition() + dataOffset;

98.
99.                 //If there is a non-
    zero value than it should indicate a pointer to the security cookie
100.                if(LittleEndian.getUInt(bytes, foa+0x3C) > 0){
101.                    return true;
102.                }
103.            }
104.            return false;
105.        }
106.
107.        @Override
108.        public String getCodeSegmentName() {
109.            return codeSegmentName;
110.        }
111.
112.        @Override
113.        public byte[] getCode()
114.            {
115.                return code;
116.            }
117.
118.        @Override
119.        public void setBytes(byte[] bytes)
120.            {
121.                super.setBytes(bytes);
122.                load();
123.            }
124.
125.        @Override
126.        public CODE_TYPE getType()
127.            {
128.                ImageFileHeader ifx = fileHeader.getFileHeader();
129.
130.                if (ifx.getMachine() == 0x01c4)
131.                    {
132.                        return CODE_TYPE.ARM;
133.                    }
134.                return CODE_TYPE.x86;
135.            }
```

`

```
136.
137.         @Override
138.         public int getBaseAddress()
139.             {
140.                 return (int) fileHeader.getImageOptionalHeader().getImageBase() + codeO
    ffset;
141.             }
142.
143.         @Override
144.         public int getStartAddress()
145.             {
146.
147.                 return (int) fileHeader.getImageOptionalHeader().getImageBase() + fileH
    eader.getImageOptionalHeader().getAddressOfEntryPoint();
148.             }
149.
150.     }
```

X86.java

```
1.  package edu.millermj.disassembler;
2.
3.  import java.util.LinkedList;
4.  import java.util.TreeMap;
5.  import java.util.TreeSet;
6.
7.  import capstone.Capstone.CsInsn;
8.  import edu.millermj.x86.x86Address;
9.  import edu.millermj.x86.x86AddressVisitor;
10. import edu.millermj.x86.x86AddressVisitorPrint;
11. import edu.millermj.x86.x86Call;
12. import edu.millermj.x86.x86Jump;
13. import edu.millermj.x86.x86JumpCond;
14. import edu.millermj.x86.x86Normal;
15. import edu.millermj.x86.x86Ret;
16. import edu.millermj.x86.x86VisitorIns;
17.
18. public class x86 extends DisassembleInterface
19.     {
20.         private LinkedList<Integer>   tempList    = new LinkedList<Integer>();
21.         byte[]                        data;
22.         Integer                       start;
23.         TreeSet<Integer>            done        = new TreeSet<Integer>();
24.         byte[]                        code        = new byte[20];
25.
26.         /**
27.          * Get Ins
28.          *
29.          */
30.         private Ins getIns(int current)
31.             {
32.                 Ins i = null;
33.                 Integer startAddressOffset = (current - start);
34.                 if (startAddressOffset < data.length)
35.                     {
36.                         if (!done.contains(startAddressOffset) && startAddressOffset >=
     0)
37.                             {
38.                                 done.add(startAddressOffset);
```

`

```
                                for (int dataAddress = startAddressOffset, codeAddress
    = 0;
                                        dataAddress < startAddressOffset + 20; dataAddr
    ess++, codeAddress++)
                                    {
                                        if( dataAddress < data.length) {
                                            code[codeAddress] = data[dataAddress];
                                        }
                                    }
                                CsInsn instruct = cs.disasm(code, current, 1)[0];
                                i = new Ins(instruct, current);
                                String mnemonic = instruct.mnemonic;
                                int size = instruct.getSize();
                                i.setSize(size);
                                if (instruct == null || mnemonic == null)
                                    {
                                        i = null;
                                    }
                            }
                        }
                    else
                        {
                            System.err.println(startAddressOffset);
                        }

                    return i;
                }

            private x86Address getAddressItem(Ins i, int current)
                {
                    x86Address result = null;
                    Boolean branch = true;
                    int next = current + i.getSize();
                    switch (i.getMnemonic())
                        {
                        case "call":

                            x86Call xCall = new x86Call(i, next);
                            result = xCall;
                            break;
                        case "jmp":
                            x86Jump j = new x86Jump(i, next);
                            result = j;
                            break;
                        case "jz":
                        case "jnz":
                        case "ja":
                        case "jae":
                        case "je":
                        case "jne":
                        case "jg":
                        case "jge":
                        case "jle":
                        case "jl":
                        case "js":
                        case "jb":
                        case "jbe":
                        case "jns":
                            x86JumpCond jc = new x86JumpCond(i, next);
                            result = jc;
```

`

```
97.                         break;
98.                     case "ret":
99.                         result = new x86Ret(i, next);
100.                        break;
101.                    default:
102.                        result = new x86Normal(i, next);
103.                        branch = false;
104.                        break;
105.                    }
106.                if (result != null)
107.                    {
108.                        i.setBrach(branch);
109.                    }
110.                return result;
111.            }
112.
113.        @Override
114.        public Code disassemble(LoaderInterface l)
115.            {
116.                TreeSet<x86VisitorIns> instructs = new TreeSet<x86VisitorIns>();
117.                Code c = new Code();
118.                data = l.getCode();
119.                done.clear();
120.                TreeMap<Integer, Ins> resultMap = c.getCodeMap();
121.                start = l.getBaseAddress();
122.                TreeSet<Integer> list = new TreeSet<Integer>();
123.                list.add(l.getStartAddress());
124.                Ins i;
125.                Integer current;
126.                x86AddressVisitor visitor = new x86AddressVisitor(tempList);
127.                while (!list.isEmpty())
128.                    {
129.                        current = list.first();
130.                        list.remove(current);
131.                        tempList.clear();
132.                        i = getIns(current);
133.                        if (i != null)
134.                            {
135.                                resultMap.put(current, i);
136.                                x86Address x = getAddressItem(i, current);
137.                                if (x != null)
138.                                    {
139.                                        instructs.add((x86VisitorIns) x);
140.                                        x.accept(visitor);
141.                                        for (Integer k : tempList)
142.                                            {
143.                                                if (!done.contains(k))
144.                                                    {
145.                                                        list.add(k);
146.                                                    }
147.                                            }
148.                                    }
149.                            }
150.                    }
151.                x86AddressVisitorPrint p = new x86AddressVisitorPrint();
152.                for (x86VisitorIns in : instructs)
153.                    {
154.                        in.accept(p);
155.                    }
156.                return c;
```

`

```
157.             }
158.
159.         @Override
160.         public void generateFunctions(Code c)
161.             {
162.                 TreeSet<Ins> instructions = new TreeSet<Ins>();
163.                 c.getCodeMap().forEach((k, v) -> {
164.
165.                     if (v.getMnemonic().equals("call"))
166.                         {
167.                             if (!instructions.contains(v))
168.                                 {
169.                                     Long callL = v.getIMMAddress();
170.
171.                                     if (callL != null)
172.                                         {
173.                                             Integer callAddress = callL.intValue();
174.                                             if (c.getCodeMap().containsKey(callAddress)
    )
175.                                                 {
176.                                                     instructions.add(c.getCodeMap().get
    (callAddress));
177.                                                 }
178.                                         }
179.                                 }
180.                         }
181.                 });
182.                 instructions.forEach((v) -> {
183.                     try
184.                         {
185.                             Ins next = instructions.higher(v);
186.                             if (next != null)
187.                                 {
188.                                     Function f = new Function(v, next);
189.                                     c.getFunctionMap().put(v.getAddress(), f);
190.                                     Integer startAddress = v.getAddress();
191.
192.                                     f.addEntryPoint(v);
193.
194.                                     while (next.getAddress() > startAddress)
195.                                         {
196.                                             if (v.getMnemonic().equals("ret"))
197.                                                 {
198.                                                     f.addExitPoint(v);
199.                                                 }
200.                                             startAddress = c.getCodeMap().higherKey(v.g
    etAddress());// get next ins
201.                                             v = c.getCodeMap().get(startAddress);
202.                                         }
203.                                 }
204.                         }
205.                     catch (Exception e)
206.                         {
207.                             e.printStackTrace();
208.                         }
209.                 });
210.             }
211.
212.     }
```

`

Import.java

```
1.  package application.database;
2.
3.  import java.io.File;
4.  import java.io.FileInputStream;
5.  import java.sql.ResultSet;
6.  import java.sql.SQLException;
7.  import java.sql.Statement;
8.  import java.time.LocalDate;
9.  import java.util.ArrayList;
10. import java.util.Iterator;
11. import java.util.List;
12.
13. import javax.sql.rowset.serial.SerialBlob;
14.
15. import capstone.Capstone;
16. import gui.ShortenedFunction;
17. import gui.ShortenedIns;
18. import javafx.beans.property.DoubleProperty;
19. import javafx.beans.property.IntegerProperty;
20. import javafx.beans.property.SimpleDoubleProperty;
21. import javafx.beans.property.SimpleIntegerProperty;
22. import javafx.beans.property.SimpleStringProperty;
23. import javafx.beans.property.StringProperty;
24. import javafx.collections.FXCollections;
25. import javafx.collections.ObservableList;
26.
27. public class Import {
28.
29.     private IntegerProperty id = new SimpleIntegerProperty();
30.     private StringProperty md5_hash = new SimpleStringProperty();
31.     private DoubleProperty file_size = new SimpleDoubleProperty();
32.     private LocalDate date_created;
33.     private StringProperty dateCreated = new SimpleStringProperty();
34.     private StringProperty name = new SimpleStringProperty();
35.
36.     private byte[] originalBytes;
37.     private List<ShortenedFunction> functions = new ArrayList<ShortenedFunction>();
38.     private List<ShortenedIns> instructions = new ArrayList<ShortenedIns>();
39.
40.     private Characteristics importCharacteristics;
41.
42.     //#FIXME: not public
43.     public section code_section;
44.
45.     private int project_id = 0;
46.
47.     private String insert = "INSERT INTO import (md5_hash, file_size, name) VALUES (?,?
    ,?)";
48.     private String update = "UPDATE import SET md5_hash = ?, file_size = ?, name = ? WH
    ERE id = ?";
49.     private String insert_pivot = "INSERT INTO project_imports (project_id, import_id)
    VALUES (?,?)";
50.     private String insert_raw_bytes = "INSERT INTO original_binary (import_id, content)
     VALUES (?,?)";
51.     private String insert_function = "INSERT INTO functions (import_id, start_address,
    end_address, alias) VALUES (?,?,?,?)";
52.     private String insert_instruction = "INSERT INTO instructions (import_id, address,
    mnemonic, operands, opcodes, opsize) VALUES (?,?,?,?,?,?)";
```

`

```
53.     private static String select_by_project = "SELECT import.name, import.id, import.da
    te_imported FROM project_imports " +
54.                                           "INNER JOIN import ON import.id = project_impor
    ts.import_id " +
55.                                           "WHERE project_imports.project_id = ? ";
56.
57.     private String select_by_id = "SELECT name, date_imported, file_size FROM import WH
    ERE id = ?";
58.     private String select_functions = "SELECT id, start_address, end_address, alias FRO
    M functions WHERE import_id = ?";
59.     private String select_instructions = "SELECT id, address, mnemonic, operands, opcod
    es, comment, opsize FROM instructions WHERE import_id = ?";
60.
61.     private static String search_mnemonic = "SELECT address, mnemonic, operands, opcode
    s, opsize FROM instructions " +
62.                                             "WHERE mnemonic = ?  AND import_id = ?";
63.
64.     private static String search_mnemonic_sequence = "SELECT address, mnemonic, operand
    s, opcodes, opsize FROM instructions " +
65.                                                       "WHERE mnemonic = ?  AND import_id
     = ? AND address = ?";
66.
67.     private static String select_function_instructions = "SELECT address, mnemonic, ope
    rands, opcodes, opsize FROM instructions " +
68.                             " WHERE import_id = ? AND (address >= ? AND address < ?)";
69.
70.     /* Constructor(s) */
71.
72.     public Import() {
73.
74.     }
75.
76.     public Import(int ID) {
77.
78.         DBConnection _db = null;
79.
80.         try {
81.             _db = new DBConnection();
82.
83.             this.setId(ID);
84.
85.             //populate the object
86.             java.sql.PreparedStatement prepared_stmt = _db.conn.prepareStatement(select
    _by_id);
87.             prepared_stmt.setInt(1, ID);
88.
89.             ResultSet rs = prepared_stmt.executeQuery();
90.
91.             while(rs.next()){
92.                 this.setName(rs.getString("name"));
93.                 this.setDateCreated(rs.getString("date_imported"));
94.                 this.date_created = rs.getDate("date_imported").toLocalDate();
95.             }
96.
97.             //first get the functions
98.             prepared_stmt = _db.conn.prepareStatement(select_functions);
99.             prepared_stmt.setInt(1, ID);
100.
101.             rs = prepared_stmt.executeQuery();
102.
```

`

```
103.            while(rs.next()) {
104.
105.                functions.add(new ShortenedFunction(rs.getInt("start_address"), rs.getI
    nt("end_address"), rs.getString("alias")));
106.            }
107.
108.            //Now get the instructions
109.            prepared_stmt = _db.conn.prepareStatement(select_instructions);
110.            prepared_stmt.setInt(1, this.getID());
111.
112.            rs = prepared_stmt.executeQuery();
113.
114.            while(rs.next()) {
115.                instructions.add(new ShortenedIns(rs.getString("opcodes"),rs.getInt("ad
    dress"), rs.getString("mnemonic"), rs.getString("operands"), rs.getInt("opsize")));
116.            }
117.
118.            //now get the section(s)
119.            this.code_section = new section(ID);
120.
121.            //Update Characteristics
122.            this.importCharacteristics = new Characteristics(ID);
123.
124.            rs.close();
125.            _db.conn.close();
126.
127.        }catch(Exception ex) {
128.            ex.printStackTrace();
129.        }
130.    }
131.
132.    public Import(int ID, String name, LocalDate dateCreated) {
133.        this.setId(ID);
134.        this.setName(name);
135.        this.setDateCreated(dateCreated.toString());
136.        this.date_created = dateCreated;
137.
138.        DBConnection _db = null;
139.
140.        try {
141.            _db = new DBConnection();
142.
143.            //first get the functions
144.            java.sql.PreparedStatement prepared_stmt = _db.conn.prepareStatement(select
    _functions);
145.            prepared_stmt.setInt(1, ID);
146.
147.            ResultSet rs = prepared_stmt.executeQuery();
148.
149.            while(rs.next()) {
150.
151.                functions.add(new ShortenedFunction(rs.getInt("start_address"), rs.getI
    nt("end_address"), rs.getString("alias")));
152.
153.            }
154.
155.            //Now get the instructions
156.            prepared_stmt = _db.conn.prepareStatement(select_instructions);
157.            prepared_stmt.setInt(1, this.getID());
158.
```

`

```
159.            rs = prepared_stmt.executeQuery();
160.
161.            while(rs.next()) {
162.                instructions.add(new ShortenedIns(rs.getString("opcodes"),rs.getInt("ad
    dress"), rs.getString("mnemonic"), rs.getString("operands"), rs.getInt("opsize")));
163.            }
164.
165.            rs.close();
166.
167.        }catch(Exception ex) {
168.
169.        } finally {
170.            try {
171.                _db.conn.close();
172.            } catch (SQLException e) {
173.                // TODO Auto-generated catch block
174.                e.printStackTrace();
175.            }
176.        }
177.    }
178.
179.    public void Save() {
180.        DBConnection _db = new DBConnection();
181.
182.        try {
183.
184.            if(this.getID() > 0) { //update
185.
186.                java.sql.PreparedStatement prepared_stmt = _db.conn.prepareStatement(up
    date);
187.                prepared_stmt.setString(1, this.getMD5Hash());
188.                prepared_stmt.setDouble(2, this.getFileSize());
189.                prepared_stmt.setString(3, this.getName());
190.                prepared_stmt.setInt(4, this.getID());
191.
192.                prepared_stmt.execute();
193.
194.            } else { // insert
195.
196.                java.sql.PreparedStatement prepared_stmt = _db.conn.prepareStatement(in
    sert,Statement.RETURN_GENERATED_KEYS);
197.
198.                prepared_stmt.setString(1, this.getMD5Hash());
199.                prepared_stmt.setDouble(2, this.getFileSize());
200.                prepared_stmt.setString(3, this.getName());
201.
202.                int affectedRows = prepared_stmt.executeUpdate();
203.
204.                if(affectedRows == 0) {
205.                    throw new SQLException("Failed Creating Import");
206.                }
207.
208.                try (ResultSet generated_key = prepared_stmt.getGeneratedKeys()) {
209.                    if(generated_key.next()) {
210.                        this.setId(generated_key.getInt(1));
211.
212.                        //update the pivot table
213.                        java.sql.PreparedStatement ps_pivot = _db.conn.prepareStatement
    (insert_pivot);
214.
```

`

```
215.                         ps_pivot.setInt(1, this.getProjectID());
216.                         ps_pivot.setInt(2, this.getID());
217.
218.                         ps_pivot.execute();
219.
220.                     }else {
221.                         throw new SQLException("Failed to create import, no primary key
    ID obtained");
222.                     }
223.                 }
224.
225.                 if(importCharacteristics != null) {
226.                     System.out.println("[DEBUG] Saving Characteristics");
227.                     importCharacteristics.setImportID(this.id.getValue());
228.                     importCharacteristics.Save();
229.                 }
230.             }
231.
232.         } catch (SQLException e) {
233.             // TODO Auto-generated catch block
234.             e.printStackTrace();
235.         } catch (Exception e) {
236.             // TODO Auto-generated catch block
237.             e.printStackTrace();
238.         } finally {
239.             try {
240.                 _db.conn.close();
241.             } catch (SQLException e) {
242.                 // TODO Auto-generated catch block
243.                 e.printStackTrace();
244.             }
245.         }
246.     }
247.
248.     //#FIXME: these just be incorporated with the Save()
249.     public void SaveRawBytes(File inputFile) {
250.
251.         DBConnection _db = new DBConnection();
252.
253.         try {
254.
255.             FileInputStream fileInputStream = null;
256.
257.             this.originalBytes = new byte[(int)inputFile.length()];
258.
259.             fileInputStream = new FileInputStream(inputFile);
260.             fileInputStream.read(this.originalBytes);
261.             fileInputStream.close();
262.
263.             if(this.getID() != 0 ) {
264.
265.                 java.sql.PreparedStatement prepared_stmt = _db.conn.prepareStatement(in
    sert_raw_bytes);
266.                 prepared_stmt.setInt(1, this.getID());
267.                 prepared_stmt.setBlob(2, new SerialBlob(originalBytes));
268.
269.                 prepared_stmt.execute();
270.
271.             }
272.
```

`

```
273.         }catch(Exception e) {
274.             e.printStackTrace();
275.         }
276.     }
277.
278.     public void SaveFunctions() {
279.         DBConnection _db = new DBConnection();
280.
281.         try{
282.
283.             if(this.functions.size() > 0) {
284.
285.                 for(Iterator<ShortenedFunction> i = this.functions.iterator(); i.hasNex
    t();){
286.
287.                     ShortenedFunction f = i.next();
288.
289.                     java.sql.PreparedStatement prepared_stmt = _db.conn.prepareStatemen
    t(insert_function);
290.                     prepared_stmt.setInt(1, this.getID());
291.                     prepared_stmt.setInt(2, f.getStartAddress());
292.                     prepared_stmt.setInt(3, f.getEndAddress());
293.                     prepared_stmt.setString(4, f.getAlias());
294.
295.                     prepared_stmt.execute();
296.                 }
297.             }
298.         } catch(Exception ex) {
299.             ex.printStackTrace();
300.         }
301.     }
302.
303.     public void SaveInstructions() {
304.         DBConnection _db = new DBConnection();
305.
306.         try{
307.
308.             if(this.functions.size() > 0) {
309.
310.                 for(Iterator<ShortenedIns> i = this.instructions.iterator(); i.hasNext(
    );){
311.
312.                     ShortenedIns f = i.next();
313.
314.                     java.sql.PreparedStatement prepared_stmt = _db.conn.prepareStatemen
    t(insert_instruction);
315.                     prepared_stmt.setInt(1, this.getID());
316.                     prepared_stmt.setInt(2,f.getAddress());
317.                     prepared_stmt.setString(3, f.getMnemonic());
318.                     prepared_stmt.setString(4, f.getOperands());
319.                     prepared_stmt.setString(5, f.getOpcodeS());
320.                     prepared_stmt.setInt(6, f.getOpSize());
321.
322.                     prepared_stmt.execute();
323.                 }
324.             }
325.         } catch(Exception ex) {
326.             ex.printStackTrace();
327.         }
328.     }
```

`

```
329.
330.    public static String buildMnemonicExpression(String[] query) {
331.
332.        String operands = "";
333.
334.        if (query.length > 1) {
335.
336.            operands += " AND operands REGEXP '.*";
337.
338.            for(int i = 1; i < query.length; i++) {
339.
340.                String op = query[i].trim().replace("$", "");
341.
342.                if(op.equals("imm"))
343.                {
344.                    operands += "((0[xX][0-9a-fA-F]+)|([0-9]{1,}))";
345.                }
346.                else if(op.equals("r32"))
347.                {
348.                    operands += "[xi]{1}";
349.                }
350.                else if(op.equals("mem"))
351.                {
352.                    operands += "(ebp|esp|0x).*";
353.                }
354.                else
355.                {
356.                    operands += op;
357.                }
358.
359.                if ( i < query.length - 1 )
360.                {
361.                    operands += ",.*";
362.                }
363.            }
364.            operands += "$'";
365.
366.        }
367.        return operands;
368.    }
369.
370.    public static List<ShortenedIns> searchByMnemonicExpression(int importID, String ex
    p) {
371.
372.        List<ShortenedIns> results = new ArrayList<ShortenedIns>();
373.
374.        DBConnection _db = null;
375.
376.        try {
377.            _db = new DBConnection();
378.
379.            String[] query = exp.split("#");
380.
381.            String operands = buildMnemonicExpression(query);
382.
383.            java.sql.PreparedStatement prepared_stmt = _db.conn.prepareStatement(search
    _mnemonic + operands);
384.            prepared_stmt.setString(1, query[0]);
385.            prepared_stmt.setInt(2, importID);
386.
```

`

```
387.             ResultSet rs = prepared_stmt.executeQuery();
388.
389.             while(rs.next()) {
390.                 results.add(new ShortenedIns(rs.getString("opcodes"),rs.getInt("address
    "), rs.getString("mnemonic"), rs.getString("operands"),rs.getInt("opsize")));
391.             }
392.
393.         } catch (SQLException e) {
394.             e.printStackTrace();
395.         } finally {
396.             try {
397.                 _db.conn.close();
398.             } catch (SQLException e) {
399.                 e.printStackTrace();
400.             }
401.         }
402.         return results;
403.     }
404.
405.     public static List<ShortenedIns> searchByMnemonicSequence(int importID, String exp)
    {
406.         List<ShortenedIns> results = new ArrayList<ShortenedIns>();
407.         List<ShortenedIns> tmp = new ArrayList<ShortenedIns>();
408.
409.         DBConnection _db = null;
410.
411.         String[] expressions = exp.split("\\r\\n|\\n|\\r");
412.         int num_sequences = expressions.length;
413.         int next_address = 0;
414.
415.         String[] query = expressions[0].split("#");
416.         String operands = buildMnemonicExpression(query);
417.
418.         try {
419.             _db = new DBConnection();
420.
421.             java.sql.PreparedStatement prepared_stmt = _db.conn.prepareStatement(search
    _mnemonic + operands);
422.             prepared_stmt.setString(1, query[0]);
423.             prepared_stmt.setInt(2, importID);
424.
425.             ResultSet rs = prepared_stmt.executeQuery();
426.             List<ShortenedIns> first_ins = new ArrayList<ShortenedIns>();
427.
428.             while(rs.next())
429.             {
430.                 first_ins.add(new ShortenedIns(rs.getString("opcodes"),
431.                         rs.getInt("address"), rs.getString("mnemonic"), rs.getString("o
    perands"),
432.                         rs.getInt("opsize")));
433.             }
434.
435.             rs.close();
436.
437.             for(int z = 0; z < first_ins.size(); z++)
438.             {
439.                 ShortenedIns tmpIns = first_ins.get(z);
440.                 next_address = tmpIns.getAddress() + tmpIns.getOpSize();
441.
442.                 tmp.add(tmpIns);
```

`

```
443.
444.                 innerloop:
445.                 for(int x = 1; x < expressions.length; x++)
446.                 {
447.
448.                     String[] sub_query = expressions[x].split("#");
449.                     String[] firstInsOperands = tmpIns.getOperands().split(",");
450.
451.                     if(sub_query[1].contains("$"))
452.                     {
453.                         if(query[1].contains("$"))
454.                         {
455.                             sub_query[1] = firstInsOperands[0];
456.                         }
457.                         else if(query[2].contains("$"))
458.                         {
459.                             sub_query[1] = firstInsOperands[1];
460.                         }
461.                     }
462.                     else if (sub_query[2].contains("$"))
463.                     {
464.                         if(query[1].contains("$"))
465.                         {
466.                             sub_query[2] = firstInsOperands[0];
467.                         }
468.                         else if(query[2].contains("$"))
469.                         {
470.                             sub_query[2] = firstInsOperands[1];
471.                         }
472.                     }
473.
474.                     String sub_operands = buildMnemonicExpression(sub_query);
475.
476.                     java.sql.PreparedStatement sub_p_stmt =
477.                             _db.conn.prepareStatement(search_mnemonic_sequence + sub_op
    erands);
478.                     sub_p_stmt.setString(1, sub_query[0]);
479.                     sub_p_stmt.setInt(2, importID);
480.                     sub_p_stmt.setInt(3, (next_address));
481.
482.                     ResultSet sub_rs = sub_p_stmt.executeQuery();
483.
484.                     if(!sub_rs.isBeforeFirst()) {
485.                         break innerloop;
486.                     }
487.                     else
488.                     {
489.                         sub_rs.next();
490.
491.                         ShortenedIns sub_tmp = new ShortenedIns(sub_rs.getString("opcod
    es"),
492.                                 sub_rs.getInt("address"), sub_rs.getString("mnemonic"),
493.                                 sub_rs.getString("operands"),sub_rs.getInt("opsize"));
494.
495.                         tmp.add(sub_tmp);
496.
497.                         next_address = (next_address + sub_tmp.getOpSize());
498.                     }
```

`

```
499.                 }
500.
501.                 if(tmp.size() == num_sequences)
502.                 {
503.                     results.addAll(tmp);
504.                 }
505.                 tmp.clear();
506.             }
507.
508.         } catch (SQLException e) {
509.             e.printStackTrace();
510.         } finally {
511.             try {
512.                 _db.conn.close();
513.             } catch (SQLException e) {
514.                 e.printStackTrace();
515.             }
516.         }
517.         return results;
518.     }
519.
520.     public List<ShortenedIns> getFunctionInstructions(ShortenedFunction func) {
521.
522.         List<ShortenedIns> results =  new ArrayList<ShortenedIns>();
523.         DBConnection _db = null;
524.
525.         try {
526.
527.             _db = new DBConnection();
528.
529.             java.sql.PreparedStatement prepared_stmt = _db.conn.prepareStatement(select
    _function_instructions);
530.             prepared_stmt.setInt(1, this.getID());
531.             prepared_stmt.setInt(2,  func.getStartAddress());
532.             prepared_stmt.setInt(3, func.getEndAddress());
533.
534.             ResultSet rs = prepared_stmt.executeQuery();
535.
536.             while(rs.next()) {
537.                 //String opCodes, int address, String mnemonic, String operands
538.                 results.add(new ShortenedIns(rs.getString("opcodes"),rs.getInt("address
    "), rs.getString("mnemonic"), rs.getString("operands"),rs.getInt("opsize")));
539.             }
540.
541.         } catch (SQLException e) {
542.             // TODO Auto-generated catch block
543.             e.printStackTrace();
544.         } finally {
545.             try {
546.                 _db.conn.close();
547.             } catch (SQLException e) {
548.                 // TODO Auto-generated catch block
549.                 e.printStackTrace();
550.             }
551.         }
552.         return results;
553.     }
554.
555.     /* Static Methods */
```

`

```
556.    public static List<String> searchROPGadgets(int importID, String endingsToSearch) {
557.
558.        int depth = 5;
559.        List<String> rop_gadgets = new ArrayList<String>();
560.
561.        Import selected_import = new Import(importID);
562.
563.        ObservableList<ShortenedIns> obsInstructions = FXCollections.observableArrayLis
    t(
564.                Import.searchByMnemonicExpression(importID,endingsToSearch));
565.
566.        String[] opcodes = selected_import.code_section.content.split(" ");
567.
568.        for (int i = 0; i < obsInstructions.size(); i++) {
569.
570.            ShortenedIns si = obsInstructions.get(i);
571.
572.            int offset = si.getAddress() - 0x401000;
573.            int adjusted_offset = offset - depth;
574.
575.            List<Byte> bytes = new ArrayList<Byte>();
576.
577.            for(int j = adjusted_offset; j <= offset; j++) {
578.
579.                byte value = (byte) ((Character.digit(opcodes[j].charAt(0), 16) << 4)
580.                        + Character.digit(opcodes[j].charAt(1), 16));
581.
582.                bytes.add(value);
583.            }
584.
585.            byte[] code = new byte[bytes.size()];
586.
587.            for(int x = 0; x < bytes.size(); x++)
588.            {
589.                code[x] = bytes.get(x).byteValue();
590.            }
591.
592.            try{
593.                String tmp_gadget = "";
594.
595.                Capstone cs = new Capstone(Capstone.CS_ARCH_X86, Capstone.CS_MODE_32);
596.
597.                Capstone.CsInsn[] allInsn = cs.disasm(code, offset - depth);
598.
599.                for (int z = 0; z<allInsn.length; z++)
600.                {
601.                    tmp_gadget += allInsn[z].mnemonic + " " + allInsn[z].opStr + " # ";
602.                }
603.
604.                if (tmp_gadget.contains(endingsToSearch))
605.                {
606.                    tmp_gadget = tmp_gadget.substring(0,tmp_gadget.length() - 2);
607.
608.                    tmp_gadget = String.format("0x%08x: ", 0x401000 + (offset - depth))
    + tmp_gadget;
609.
```

`

```
610.                     rop_gadgets.add(tmp_gadget);
611.                 }
612.             }
613.             catch(Exception ex)
614.             {
615.                 System.out.println("ERROR disassembling ROP gadgets - " + ex.toString()
    );
616.             }
617.         }
618.         return rop_gadgets;
619.     }
620.
621.     public static List<ShortenedFunction> searchFunctionsAcrossImports(Project p, int i
    mportID) {
622.
623.         Import im = new Import(importID);
624.
625.         List<ShortenedFunction> funcs = im.getFunctions();
626.         List<Import> projectImports = p.imports;
627.         List<ShortenedFunction> matchingFunctions = new ArrayList<ShortenedFunction>();

628.
629.         for (int j = 0; j < projectImports.size(); j++ )
630.         {
631.             Import tmpImport = projectImports.get(j);
632.
633.             if(im.getID() != tmpImport.getID())
634.             {
635.                 for ( int i = 0; i < funcs.size(); i++)
636.                 {
637.                     List<ShortenedIns> tmpListIns = im.getFunctionInstructions(funcs.ge
    t(i));
638.
639.                     List<ShortenedFunction> tmpResults = searchImport(tmpListIns, tmpIm
    port);
640.
641.                     if (tmpResults.size() > 0)
642.                     {
643.                         matchingFunctions.addAll(tmpResults);
644.                     }
645.                 }
646.             }
647.         }
648.
649.         return matchingFunctions;
650.     }
651.
652.     public static List<ShortenedFunction> searchImport(List<ShortenedIns> sourceFuncIns
    , Import im) {
653.
654.         List<ShortenedFunction> results = new ArrayList<ShortenedFunction>();
655.         List<ShortenedFunction> funcs = im.getFunctions();
656.
657.         for(int i = 0; i < funcs.size(); i++)
658.         {
659.
660.             List<ShortenedIns> tmpListIns = im.getFunctionInstructions(funcs.get(i));

661.
662.             int seek_len = tmpListIns.size();
```

`

```
663.
664.             if (sourceFuncIns.size() == tmpListIns.size())
665.             {
666.                 int j = 0;
667.
668.                 while(j < seek_len
669.                         && tmpListIns.get(j).getMnemonic().equals(sourceFuncIns.get(j).
    getMnemonic()))
670.                 {
671.                     j++;
672.
673.                     if(j == seek_len)
674.                     {
675.                         results.add(funcs.get(i));
676.                     }
677.                 }
678.             }
679.         }
680.         return results;
681.     }
682.
683.     public static List<Import> GetAllImports(int projectID) {
684.
685.         List<Import> imports = new ArrayList<Import>();
686.         DBConnection _db = null;
687.
688.         try {
689.             _db = new DBConnection();
690.
691.             java.sql.PreparedStatement prepared_stmt = _db.conn.prepareStatement(select
    _by_project);
692.             prepared_stmt.setInt(1, projectID);
693.
694.             ResultSet rs = prepared_stmt.executeQuery();
695.
696.             while(rs.next()) {
697.                 Import i = new Import(rs.getInt("id"), rs.getString("name"), rs.getDate
    ("date_imported").toLocalDate());
698.
699.                 imports.add(i);
700.             }
701.
702.         } catch (SQLException e) {
703.             // TODO Auto-generated catch block
704.             e.printStackTrace();
705.         } finally {
706.             try {
707.                 _db.conn.close();
708.             } catch (SQLException e) {
709.                 // TODO Auto-generated catch block
710.                 e.printStackTrace();
711.             }
712.         }
713.
714.         return imports;
715.
716.     }
717.
718.     /* Getters and Setters */
719.     public IntegerProperty idProperty() {
```

`

```
720.         return this.id;
721.     }
722.
723.     public final int getID() {
724.         return id.get();
725.     }
726.
727.     public final void setId(int ID) {
728.         this.id.set(ID);
729.     }
730.
731.     public StringProperty nameProperty() {
732.         return this.name;
733.     }
734.
735.     public DoubleProperty fileSizeProperty() {
736.         return this.file_size;
737.     }
738.
739.     public final void setFileSize(long size) {
740.         this.file_size.set(size);
741.     }
742.
743.     public final Double getFileSize() {
744.         return this.file_size.get();
745.     }
746.
747.     public final String getName() {
748.         return name.get();
749.     }
750.
751.     public final void setName(String name) {
752.         this.name.set(name);
753.     }
754.
755.     public StringProperty dateCreated() {
756.         return this.dateCreated;
757.     }
758.
759.     public final String getDateCreated() {
760.         return this.dateCreated.get();
761.     }
762.
763.     public final void setDateCreated(String dateCreated) {
764.         this.dateCreated.set(dateCreated);
765.     }
766.
767.     public StringProperty md5_hash() {
768.         return this.md5_hash;
769.     }
770.
771.     public final String getMD5Hash() {
772.         return this.md5_hash.get();
773.     }
774.
775.     public final void setMD5Hash(String hash) {
776.         this.md5_hash.set(hash);
777.     }
778.
779.     public final int getProjectID () {
```

`

```
780.         return this.project_id;
781.     }
782.
783.     public final void setProjectID(int project_id) {
784.         this.project_id = project_id;
785.     }
786.
787.     public final void setOriginalBytes(byte[] bytes) {
788.         this.originalBytes = bytes;
789.     }
790.
791.     public final void setFunctions(List<ShortenedFunction> functions) {
792.         this.functions = functions;
793.     }
794.
795.     public final List<ShortenedFunction> getFunctions() {
796.         return this.functions;
797.     }
798.
799.     public final void setInstructions(List<ShortenedIns> instructions) {
800.         this.instructions = instructions;
801.     }
802.
803.     public final List<ShortenedIns> getInstructions() {
804.         return this.instructions;
805.     }
806.
807.     public Characteristics getCharacteristics() {
808.         return this.importCharacteristics;
809.     }
810.
811.     public void setCharacteristics(Characteristics c){
812.         this.importCharacteristics = c;
813.     }
814. }
```

`