Generic Reverse Engineering Architecture with Compiler and Compression Classification Components

Except where reference is made to the work of others, the work described in this dissertation is my own or was done in collaboration with my advisory committee. This dissertation does not include proprietary or classified information.

Stephen A. Torri

Certificate of Approval:

Richard Chapman Associate Professor Computer Science and Software Engineering	John A. Hamilton, Jr. Associate Professor Computer Science and Software Engineering
David Umphress Associate Professor Computer Science and Software Engineering	David Rouse Professor Fisheries & Allied Aquacultures

George T. Flowers
Dean
Graduate School

Generic Reverse Engineering Architecture with Compiler and Compression Classification Components

Stephen A. Torri

A Dissertation

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Doctor of Philosophy

Auburn, Alabama
May 9, 2009

Generic Reverse Engineering Architecture with Compiler and Compression Classification Components

Stephen A. Torri

Signature of Author

Date of Graduation

Dissertation Abstract

Generic Reverse Engineering Architecture with Compiler and Compression Classification Components

Stephen A. Torri

Doctor of Philosophy, May 9, 2009
(M.S., in Computer Science, Washington University in Saint Louis, 2004)
(B.S., in Accounting and Finance and Computer Science,
Lancaster University (UK), 2001)

195 Typed Pages

Directed by John A. Hamilton, Jr.

As more and more applications, libraries, and other types of programs are being executed in untrusted environments they will be targets of attackers. These applications are exposed to malicious programs attempting to exploit some publicly known or newly discovered vulnerability in order to produce an unwanted action. These malicious and/or suspect programs can be installed on a system without the knowledge of the user. In these circumstances reverse engineering would be able to discover the functionality of the programs without actually executing them. This is important because it is necessary to know as much about a program before executing it in a controlled environment.

Since each binary application, e.g. malicious programs for Intel X86 or Java, was produced by a compiler it would be helpful to customize the reverse engineering process by detecting which compiler was used. This research will be experimenting with methods that help detect the compiler used to create an executable program regardless of the programming language used. The method discovered for compiler detection will be added to a generic reverse engineering architecture that will utilize this information to alter the run-time behavior of the generic reverse engineering architecture.

Acknowledgments

Every great adventure has many stages and a cast of different people that contributed to it. The following people I wish to thank for their part in this journey.

To my Lord and Savior Jesus Christ because of whom none of this would be possible. It is because of and through Him I was placed here at this place in time to perform the task set before me by His power with the skills endowed to me by Him.

To my parents Joseph and Marilynn Torri without whom this journey would not have been possible. It is their unfailing belief in me and support that enabled me to complete my journey. In addition I thank my brothers, Brian and Christopher, and sister, Jennifer, for taking the time to encourage me and help me keep a right perspective on the task at hand.

To my adviser Dr John A. Hamilton, Jr. and the IA-Lab research group. I want to thank for their support, discussions and guidance on this project. Also I wish to thank the members of my committee: Dr Richard Chapman, Dr David Umphress and Dr Rouse for their contributions and their helpful discussions.

To Dr Morgan Deters for giving me the dissertation by Dr Cristina Cifuentes at Washington University in Saint Louis which inspired me to look at generic reverse engineering. Your encouragement and support helped me to see the potential in this area of research.

To Winard Britt for his support of the use of Artificial Intelligence in classification. Your comments on this dissertation, research and encouragement during this process were invaluable.

To Dr Christopher Gill who believed in my potential as a researcher and gave me a platform from which I could grow. I greatly value our discussions we had and appreciate all your assistance.

To the mentors and close friends whom I have met over the years who helped me to learn more about who I am, how I was created and to know what abilities I have been given.

Style manual or journal used Journal of Approximation Theory (together with the style known as "aums"). Bibliograpy follows van Leunen's *A Handbook for Scholars*.

Computer software used The document preparation package TEX (specifically LATEX) together with the departmental style-file `aums.sty`.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Statement of the Problem

At present there is a limited set of tools available to an organization to conduct reverse engineering work and research. Most of these tools are focused on a specific problem space with a limited set of inputs. The problem with these approaches is that they are too limited and provide no ability to reuse the solution for other problems. There is often large overlap in functionality in these approaches that cannot be reused since the functionality is so tightly coupled to the particular problem.

One purpose of this work is to contribute to the reverse engineering of software by providing a means for identifying the type of compiler used to compile a Java Class, Linux ELF or Windows PE file. Since each compiler performs its operations differently it is important to know which one produced a binary application. This information would be valuable in the reverse engineering effort. There is no method known at present for detecting the compiler used to create a binary computer application. This research shows that it is possible to detect the compiler used to create a binary computer application, regardless of the programming language used, through information gathered from the binary file headers by classifying against profiles from a list of available compilers. The research focuses on information gathered from the binary file headers and not the actual instructions executed by the system processor.

This research investigated the file properties from two different binary file types (Windows PE and Linux ELF) and an intermediate file type (Java Class) to determine the initial set of values believed to be useful to the classification of the compiler used. Once the file properties to be used were determined, an automated method was used to condense this information into training sets. A training set was produced for each set of

files created by the compilers. For example, there are three different compilers used to produce Windows PE files. Each set of Windows PE files has a training set containing the desired file properties. All of the training sets were given to an optimization method to determine the properties that are actually useful to the classification of the compiler used. This method was repeated for the Linux ELF and Java Class binary file types.

Two different optimization methods were used to optimize the neural network parameters, similar to the ideas suggested in [22] for the optimization of weights in a Feed-Forward Neural Network using a Particle Swarm Optimizer. Experiments were developed to test the effectiveness of the system on classifying unseen executable file instances, recording metrics of success rate and average distance to desired output.

The output of the optimization method was a final training set with each entry only containing the actual file properties useful for classification. In addition, the user of the optimization method receives a list of file properties used, success rate and neural network parameters. The final training set and the model parameters are incorporated into a reverse engineering architecture and demonstrated to show how the knowledge of the compiler used can alter a reverse engineering architecture's behavior.

The second contribution of this research provides a general reverse engineering architecture. The generic reverse engineering architecture contains the following elements:

1. Memory Map data structure to represent a load segment of memory.
2. File readers for the Windows PE, Linux ELF and Java class binary file formats.
3. Meta information data structure for recording discovered properties of the target binary executable and utilizing this information to alter run-time behavior.
4. Generic processing components for performing reverse engineering analysis.
5. Mechanism for flexible arrangement of discovered generic processing components.
6. XML based scheme for representing combinations of processing components.

Libreverse is a generic reverse engineering architecture created by the author that implements these elements. The approach taken in this work was to utilize the Libreverse architecture to analyze three of the most popular sets of binary file types in use today.

1.2 Motivation of the Research

This dissertation covers the areas of reverse engineering, software engineering and artificial intelligence. Reverse engineering describes a wide range of topics and is classified into two areas: software source code and executable binaries. Software source code reverse engineering focuses on the understanding of large amounts of programming instructions.

The use of reverse engineering in this area helps convert the input source code into a format (e.g. Unified Modeling Language - UML) that aids in the understanding of the software. In reverse engineering of executable binaries, the focus is on the conversion of a target binary file into a higher level language representation (e.g. C++ or Java) in order to perform actions such as source code recovery, malicious code detection and classification.

More and more applications, libraries, and other types of programs are being executed in untrusted environments. These applications are exposed to malicious programs attempting to exploit some publicly known or newly discovered vulnerability in order to produce an unwanted action. These malicious and/or suspect programs can be installed on a system without the knowledge of the user. In order to understand the functionality of the programs without actually executing them, reverse engineering can be employed to investigate the behavior of these malicious programs. Companies wishing to obtain a higher-level of assurance on their systems or to recover lost programming source code will utilize reverse engineering.

The key motivations of this research were to experiment with methods that:

- help detect properties in executable programs. The first property was the compiler used to produce Java Class, Windows PE and Linux ELF executable. The second property was the compression method used for Windows PE executable.
- produce a generic reverse engineering architecture
- utilize these detected properties to alter the run-time behavior of the generic reverse engineering architecture to eliminate the need for customized components.

1.2.1 Detecting properties in executable programs

In order to more effectively reverse engineer a program, an engineer would need to know as much information as possible about a suspect program. This information allows the engineers to customize the reverse engineering process to the suspect program, and thereby ensuring a higher degree of success than a generic tool can provide.

The Libreverse architecture is a run-time configured reverse engineering library which is designed to meet the goal of allowing the user to reverse engineer a binary program without having any prior knowledge about it. The value of adding compiler and compressed executable classification to Libreverse is that it allows the architecture to make better decisions rather than assuming one process fits all programs.

1.2.1.1 Compressed Executables

Computer systems and software have a broad impact on the lives of millions everyday. Computers have long since stopped existing as mere word-processors and gaming engines; they help run our hospitals and defense systems, as well as provide support for a significant portion of all commerce in the world. As more and more critical systems become dependent on reliable computing, the dangers of system disruptions and failures increase [1].

As the proliferation of malicious software expands, the ability to classify applications as safe or risky becomes of greater interest to both the computer security and computer forensics communities. “Risky” applications can arise from either deliberate

attempts to do ill-intent (as in viruses, worms, and the like) or merely poor software development (poor error-checking, use of risky functions, etc). Taking into account only software created with malicious intent, the number of known viruses today has skyrocketed into the tens of thousands. The number of identified malicious programs is surely only a subset of all viruses in the wild [56]. Viruses are also increasing in sophistication, including methods of encryption, compression, and mutation in order to hide their malicious intent. Utilizing these methodologies makes identifying malware with traditional signature-based techniques more difficult, since self-modification can change the signature of the application. Research has shown that virus variants altered by mutation, encryption, and obfuscation prove more difficult to identify by many commercial malware detectors [12].

This work seeks to contribute to reverse engineering research for identifying malicious applications through static analysis. Specifically, the classification program attempts to identify if an application has capabilities for self-modification (typically encryption and compression), which can, in turn, make reverse-engineering the application easier. Further, if the capability for compression exists, the system attempts to classify, based on a number of the properties of the executable, which type of compression was used. This approach can be advantageous in practice, since it does not require actually executing potentially dangerous software in order to analyze it. Further, the approach does not require the source code of the application to be analyzed, since source code is not readily available for analysis in a great deal of software (even when considering fairly optimistic predictions made by open source enthusiasts [68]).

In order to perform the analysis, first, information is extracted from the Windows Portable Executable (PE) headers [39] from many common Windows executable applications. Then, a General Regression Neural Network (GRNN) [53] learns from the training data and attempts to estimate the likelihood of compression of a previously unseen application. Once a training set consisting of file summaries is constructed along with the general regression neural network, two different optimization methods are used to optimize the neural network parameters, similar to the ideas suggested in [22] for

the optimization of weights in a Feed-Forward Neural Network using a Particle Swarm Optimizer. Experiments are designed to test the effectiveness of the system on classifying unseen application instances, recording metrics of success rate and average distance to desired output. The structure of the training set and the network parameters are adjusted for the more difficult problem of identifying which of three popular modern compression utilities was used to compress a given application.

1.2.1.2 Is there a situation beyond reverse engineering where compiler and compressed executable detection would be useful?

Consider the situation where a company wants to install a particular application on a server that will be used for secure transfer and manipulation of private customer health records. The company wants to be assured that the application risks are known and their impact is minimized. In order to do this, the company would need a tool that could identify which compiler was used to produce the binary computer application. This information would enable the company to investigate the compiler for any defects known to be security vulnerabilities. They may want to know if the compiler uses any standard functions known to be insecure. These functions could be used in the application either directly linked into the application or through the use of a shared library. If the company has this information it would take appropriate action to mitigate the risks.

A second situation where compiler classification is important is when attempting to reverse engineer binary programs. In compilation, certain semantic information is lost transforming the programming instructions from a certain language into an intermediate language like Single Static Assignment (SSA) [17] or Register Transfer Language (RTL) [49]. Various compilers use one or more of the intermediate languages when compiling applications. So in order to decompile an executable a researcher would need to know which one of the compilers was used to create the program. This information would be used to customize their approach. For example, GNU Compiler Collection (GCC) was originally designed to use RTL as its intermediate language. This was true

for compilers created before version 4.0. After 4.0 GCC used SSA for representing the instructions during the optimization phase before being translated into RTL for the final conversion to binary instructions. Therefore if researchers found out that the compiler used was GCC 3.3 they would know that they would have to customize the process to recover the semantic information lost from conversion to RTL. Otherwise if the compiler were GCC 4.2 or later then they would know they would have to deal with the translation to SSA and then to RTL.

Another situation is where a company is attempting to recover the source code for a legacy application. The company is unsure if the application was protected with any a compression utility. In order to be assured that the instructions contained in the legacy application are not the decompression instructions the company would first use a tool to detect which compression utility, if any, was applied to it. If no compression utility was detected then the company could be assured the instructions are from the legacy application. Otherwise, the company would have to use another tool to decompress the legacy application in order to recover the original binary instructions.

1.2.1.3 How can machine learning be used for classifying?

Given the evidence of the past successes in the area of artificial intelligence (AI) methods being applied to computer defense [7], this work seeks to take advantage of machine learning algorithms (a subset of AI). Neural networks have been applied to a large array of classification and regression problems. They also have been shown to be capable of generalizing from a small, representative training set [28]. For example, in [58], a neural network was implemented to classify and prevent boot-disk viruses. Neural networks have also proven useful for SPAM detection and filtering [13], for analyzing user behavior for anomalies [57], and in intrusion detection [31].

1.2.2 Generic reverse engineering architecture

While custom solutions to specific problems typically perform very well, they fail to be useful beyond the scope of the original problem. Generic reverse engineering systems

are necessary for reverse engineering source code files [30][9] and binary applications [14] to aid in software quality assurance, code review and understanding of complex software systems. This work seeks to construct a generic reverse engineering architecture in a way that allows for rearrangement of components in a flexible manner. This means components of the generic reverse engineering architecture must be designed to allow for information to be passed without knowledge of the producer or consumer. Also each component must be designed to perform a fundamental operation that cannot be reduced any further.

1.2.3 Modification of the run-time behavior of generic reverse engineering architectures

Given the problems with specific reverse engineering solutions, this work seeks to construct a mechanism which will allow the transfer of secondary information, called meta information, between components. The meta information will be used be each component to alter its behavior. For example meta information will allow a component to perform algorithm X when the X meta information is present otherwise a default algorithm is utilized.

1.3 Summary

There are a limited set of tools available to an organization to conduct reverse engineering work and research. What is needed is a generic reverse engineering tool, used in research and production, that can use detected properties from the binary applications to make produce better results than assuming one process fits all programs.

Chapter 2

Background and Literature Review

The purpose of this chapter is to provide background information and literature review on the different uses of reverse engineering. Some of the techniques used in compiling programming languages to machine instructions use the same data structures as the reverse engineering process. Therefore, it is necessary to review techniques used in both.

2.1 Reverse Engineering Definitions

- **Compiler**: A computer application that transforms instructions written in a programming language into instructions for a target platform architecture.
- **Decompiler**: A binary computer application that performs the decompiling of a target binary computer application.
- **Decompiling**: The act of transforming instructions for a target platform architecture into the high-level representation of a programming language.
- **Disassembler**: A binary computer application that performs the transforming of executable instructions for a target binary computer application into an assembly language. The assembly language used depends on the CPU architecture of the binary computer applications. (E.g. x86 binary disassembler produces x86 assembly language)
- **DOS**: Disk operating system for personal computers. One example is MS-DOS. A 16-bit operating system released by Microsoft.
- **Executable Binary**: A computer application stored in a binary format of an operating system used by a user for a particular task.

- **Reverse engineering**: A standard definition for reverse engineering was given by Chikofsky and Cross [11] in 1990:

  > Reverse engineering is the process of analyzing a subject [software] system to:
  >
  > – identify the systems components and their interrelationships and
  >
  > – create representations of the system in another form or at a higher level of abstraction

- **Self-modifying Programs**: An executable binary that hides the actual instructions that will be executed through encryption and/or compression.

- **XML**: Extensible Markup Language (XML) is a simple, very flexible text format derived from SGML (ISO 8879). Originally designed to meet the challenges of large-scale electronic publishing, XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere [65].

2.2 Background

In order to fully understand how the compressed executable and compiler classification research use the information in binary file format headers it is necessary to explain the different file formats.

2.2.1 Windows Portable Executable (PE)

The PE format is the main executable format for binary computer applications for the Windows operating system. It maintains backwards compatibility with MS-DOS for software applications. For the sake of brevity only the DOS, PE, optional and section headers are described.

As can be seen in Figure 2.1 and Table 2.1, a PE file requires overhead to allow it to be portable amongst the different versions of Windows. The first section is always a DOS header, often referred to as the DOS stub, which is a small application that runs

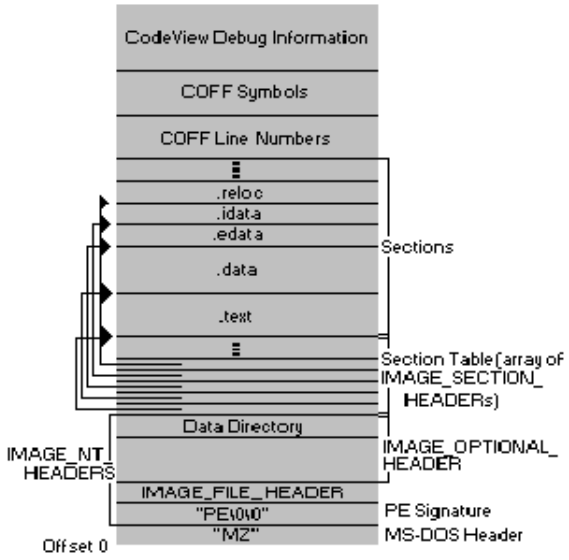


Figure 2.1: PE File Header

1										2										3										1
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
DOS 2.0 Header (Includes base of the PE image header)																															
COFF File Header																															
Section Headers																															

Table 2.1: Typical Portable EXE File Layout

under MS-DOS and prints the message: "This program cannot be run in DOS mode." Normally, having such a thing at the beginning of an executable would likely cause the application to fail to run. However, as stated in the official documentation of the PE format, the file offset to the PE signature is placed at location 0x3c within the stub, which allows the image to be run by Windows even with the stub present. [39]

2.2.1.1 DOS Header

The DOS header, in most cases, is only included in an executable program to provide the MS-DOS signature and the location of the PE header. The signature of a DOS header is MZ which is used by the Windows loader to know it is at least dealing

1																2																3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Machine																NumberOfSections															
TimeDateStamp																															
PointerToSymbolTable																															
NumberOfSymbols																															
SizeOfOptionalHeader																Characteristics															

Table 2.2: Fields of the PE COFF File Header

with a DOS program. In most versions of Microsoft Windows (e.g. Vista, XP, 2000 Professional) today the loader skips to the location in the binary file where the PE header begins. If the program was executed in DOS mode, a string is printed saying that the application cannot be executed in this mode.

2.2.1.2 PE Header

2.2.1.2.1 Common Object File Format (COFF) File Header : The PE COFF File Header, as shown in Table 2.2 is composed of a signature and the following fields. The signature is used to identify the kind of Windows file. In previous versions of Windows 3.0 and 3.1 operating systems the signature was marked by the four-bytes "NE00", but the most recent version of Windows (e.g. Vista, XP, NT), the signature is marked by the string "PE00". The following list are COFF header properties [39] that could be used during the classification process:

- **Machine**: The number that identifies the type of target machine (e.g. x86, PowerPC)
- **NumberOfSections**: The number of sections. This indicates the size of the section table, which immediately follows the headers.
- **TimeDateStamp**: The low 32 bits of the number of seconds since 00:00 January 1, 1970 (a C run-time time_t value), that indicates when the file was created.
- **PointerToSymbolTable**: The file offset of the COFF symbol table, or zero if no COFF symbol table is present. [39]

- **NumberOfSymbols**: The number of entries in the symbol table. This data can be used to locate the string table, which immediately follows the symbol table.
- **SizeOfOptionalHeader**: The size of the optional header, which is required for executable files but not for object files.
- **Characteristics**: The flags that indicate the attributes of the file.

The fields of interest are the number of sections and the characteristics. There should be at least two sections in a file to denote the code and data sections of a binary computer application. Not all compilers follow the same arrangement of sections. The number of sections lets the security researcher know at least how many sections are contained in PE file. The characteristics field is there to let the Windows operating system know if it is reading in an EXE versus a DLL.

2.2.1.2.2 Optional Header : There are many fields in the optional header, as seen in Figure 2.3, that give information on the kind of binary program contained in a Windows PE file. The first indication is the Magic field. This tells the Windows binary loader that it is dealing with a 32-bit version (PE) or a 64-bit version (PE+) Windows program. This information cannot be falsified in a system because there is a slight difference in how certain fields would be interpreted by the Windows binary loader. For example the BaseOfData field is present in the Windows PE (32-bit version) but not in the Windows PE+ (64-bit version). Falsifying the Magic field will cause the Windows loader to improperly read the binary computer application from the system disk possibly crashing the Windows loader. There are only two accepted hex values stated in the PE specification [39] of '0x10b' for Windows PE and '0x20b' for Windows PE+. The following are Optional Header properties [39] that could be used during the classification process:

- **Magic**: The unsigned integer that identifies the state of the image file. (0x10B = PE), (0x107 = ROM image), (0x20B = PE32+)

1																2																3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Magic																MajorLinkerVersion								MinorLinkerVersion							
SizeOfCode																															
SizeOfInitializedData																															
SizeOfUninitializedData																															
AddressOfEntryPoint																															
BaseOfCode																															
BaseOfData																															

Table 2.3: Fields of the PE Optional Header

- **MajorLinkerVersion**: The linker major version number.
- **MinorLinkerVersion**: The linker minor version number.
- **SizeOfCode**: The size of the code (text) section, or the sum of all code sections if there are multiple sections.
- **SizeOfInitializedData**: The size of the initialized data section, or the sum of all such sections if there are multiple data sections.
- **SizeOfUninitializedData**: The size of the uninitialized data section (marked by the section name BSS), or the sum of all such sections if there are multiple BSS sections.
- **AddressOfEntryPoint**: The address of the entry point relative to the image base when the executable file is loaded into memory. For program images, this is the starting address. For device drivers, this is the address of the initialization function. An entry point is optional for DLLs. When no entry point is present, this field must be zero.

The only remaining fields in the optional header that are of interest are the table pointer as seen in Table 2.4 and described in the list below [39]. The table pointers give the relative virtual address in where each table is located in memory and its size. These tables are used by the Windows binary loader to prepare the binary program for execution. There is one table in particular of interest called the Import Table. This

1										2										3									
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Export Table																															
Import Table																															
Resource Table																															
Exception Table																															
Certificate Table																															
Base Relocation Table																															
Debug																															
Architecture																															
Export Table																															
Global Pointer																															
TLS Table																															
Load Config Table																															
Bound Import																															
IAT																															
Delay Import Descriptor																															
CLR Runtime Header																															
Reserved																															

Table 2.4: PE Optional Header Directories

table tells what DLLs are needed to run the binary program plus what functions in them are accessed.

It is important to note that while these tables are listed in the optional header there does not need to be a matching section header for each one. What can happen is that certain tables, like the import table, can be packed into another section. So the only way to be sure of the presence of the Import Table in a Windows PE file is to first use the Import Table pointer and size if they are non-zero values. If either is a zero value then a search of the section headers is required to find the Import Table. The tables listed in the Optional Header may provide details useful to classifying a certain property but certain tools may desire to hide them from analysis.

- **Export Table**: The export table address and size. This table describes the exported symbols that can be used through dynamic linking.
- **Import Table**: The import table address and size. This table describes the imported symbols from other dynamically linked libraries.

- **Resource Table**: The resource table address and size. The resoure table is a tree base data structure describing information used by the application like icons and version information.

- **Exception Table**: The exception table address and size. The exception table contains a list of function table entries used for exception handling.

- **Certificate Table**: The attribute certificate table address and size. The certificate table contains digital certificates, like Authenticode signatures [40], which are used by a verification policy, e.g. Win32 WinVerifyTrust function [40], to determine the origin and integrity of an application.

- **Base Relocation Table**: The base relocation table address and size. The base relocation table is used to adjust base relocations if the operating system loader cannot load the image at the desired location stated in the PE header.

- **Debug**: The debug data starting address and size. The debug data contains all the compiler generated debug information.

- **Architecture**: Reserved for future use, must be 0.

- **Global Pointer**: The RVA of the value to be stored in the global pointer register. The size member of this structure must be set to zero. According to Pietrek [47] the global pointer is "a preassigned value for accessing data within a load module". Further Pietrek says that "On the IA64, each instruction is 41 bits in length. As such, it's impossible for a 41-bit IA64 instruction to contain a complete 64-bit address. Instead, all memory accesses must be done by using a 64-bit pointer value loaded into a general-purpose register. When working with data defined within a load module, the global pointer makes it possible to put the target address that you want into a register using a single instruction."

- **TLS Table**: The thread local storage (TLS) table address and size. The TLS table "provides direct PE and COFF support for static thread local storage (TLS).

TLS is a special storage class that Windows supports in which a data object is not an automatic (stack) variable, yet is local to each individual thread that runs the code. Thus, each thread can maintain a different value for a variable declared by using TLS." [39]

- **Load Config Table**: The load configuration table address and size. The load configuration table was used to describe information too big to be described in the file header but it is presently used by Windows XP and later version to contain structured exception handlers.

- **Bound Import**: The bound import table address and size. The bound import table contains a list of precomputed real addresses for each of the imported functions used from DLLs [37].

- **Delay Import Address Table (IAT)**: The import address table (IAT) address and size. The IAT is used to support a uniform mechanism for applications to delay loading of a DLL until the first call into that DLL.

- **Delay Import Descriptor**: The delay import descriptor address and size. Allows access to the delay IAT to update the entry point function pointers that reside in the data section of the image and initially refer to the delay load thunks. A thunk is a mechanism which allows a 16-bit application to access 32-bit DLLs and vice versa. [38].

- **Common Language Runtime (CLR) Runtime Header**: The CLR run time header address and size. The CLR is used to indicate that a object file contains managed code.

- **Reserved**: must be zero [39]

2.2.1.2.3 Section Header : Each section header, as shown in Table 2.5, has a name that is supposed to be descriptive of the contents of the section. The section header information was derived from [39]. While there are predefined traditional names

1										2																				3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
Name																															
VirtualSize																				VirtualAddress										
SizeOfRawData																				PointerToRawData										
PointerToRelocations																				PointerToLineNumbers										
NumberOf Relocations										NumberOf LineNumbers										Characteristics										

Table 2.5: PE Section Header

for some sections there is no guidance as to what happens in the Windows binary loader when it deals with the names that deviate from the norm. In some suspected malicious files some utilities will create sections with names specific to the utility. For example the PEtite program sometimes makes a section header which contains the decrypter titled as ".petite". Now this is a good indication that the binary program has been compressed with the PEtite program, but this has to be confirmed by other means. It is necessary to not assume anything about an application but to confirm each property. Just labeling a section as ".petite" does not mean the PEtite program was used to compressed the application. The following list of section header information [39] can be used for classification:

- **Name**: An 8-byte, null-padded UTF-8 encoded string. If the string is exactly 8 characters long, there is no terminating null. For longer names, this field contains a slash (/) that is followed by an ASCII representation of a decimal number that is an offset into the string table. Executable images do not use a string table and do not support section names longer than 8 characters. Long names in object files are truncated if they are written to an executable file.

- **VirtualSize**: The total size of the section when loaded into memory. If this value is greater than SizeOfRawData, the section is zero-padded. This field is valid only for executable images and should be set to zero for object files.

- **VirtualAddress**: For executable images, the address of the first byte of the section relative to the image base when the section is loaded into memory. For

object files, this field is the address of the first byte before relocation is applied; for simplicity, compilers should set this to zero. Otherwise, it is an arbitrary value that is subtracted from offsets during relocation.

- **SizeOfRawData**: The size of the section (for object files) or the size of the initialized data on disk (for image files). For executable images, this must be a multiple of FileAlignment from the optional header. If this is less than VirtualSize, the remainder of the section is zero-filled. Because the SizeOfRawData field is rounded but the VirtualSize field is not, it is possible for SizeOfRawData to be greater than VirtualSize as well. When a section contains only uninitialized data, this field should be zero.

- **PointerToRawData**: The file pointer to the first page of the section within the COFF file. For executable images, this must be a multiple of FileAlignment from the optional header. For object files, the value should be aligned on a 4byte boundary for best performance. When a section contains only uninitialized data, this field should be zero.

- **PointerToRelocations**: The file pointer to the beginning of relocation entries for the section. This is set to zero for executable images or if there are no relocations.

- **PointerToLinenumbers**: The file pointer to the beginning of line-number entries for the section. This is set to zero if there are no COFF line numbers. This value should be zero for an image because COFF debugging information is deprecated.

- **NumberOfRelocations**: The number of relocation entries for the section. This is set to zero for executable images.

- **NumberOfLinenumbers**: The number of line-number entries for the section. This value should be zero for an image because COFF debugging information is deprecated.

- **Characteristics**: The flags that describe the characteristics of the section.

1																																
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3
e_ident																																
e_type																e_machine																
e_machine																																
e_version																																
e_entry																																
e_phoff																																
e_shoff																																
e_flags																																
e_ehsize																e_phentsize																
e_phnum																e_shentsize																
e_shnum																e_shstrndx																

Table 2.6: ELF Header

2.2.2 Linux ELF

The Linux kernel has had three variations for its binary file format over the years. The Executable and Linking Format (ELF) [62] format is the present generation binary file format used for all Linux binary programs For the sake of brevity only the ELF header, program header table and the section headers are covered.

2.2.2.1 ELF Header

The ELF header, as shown in Table 2.6, starts with a unique identifier. The unique identifier is read by the Linux binary loader to confirm it is executing an ELF program. It will also read properties from the ELF header like which processor it was intended to run and where to start reading the program header and the section headers. These properties allow the binary loader to confirm the ELF program is capable of being executed on the system (e.g. x86 ELF program executed on a x86 linux host) and how to prepare the ELF program's image in system memory.

- **e_ident**: The initial bytes mark the file as an object file and provide machine-independent data with which to decode and interpret the file's contents.
- **e_type**: This identifies the object file type.
- **e_machine**: This value specifies the required architecture for an individual file.

- **e_version**: This identifies the object file version.
- **e_entry**: This gives the virtual address to which the system first transfers control, thus starting the process.
- **e_phoff**: This holds the program header table's file offset in bytes.
- **e_shoff**: This holds the section header table's file offset in bytes.
- **e_flags**: This holds processor-specific flags associated with the file.
- **e_ehsize**: This holds the ELF header's size in bytes.
- **e_phentsize**: This holds the size in bytes of one entry in the file's program header table. All entries are the same size.
- **e_phnum**: This holds the number of entries in the program header table. Thus, the product of e_phentsize and e_phnum gives the table's size in bytes.
- **e_shentsize**: This holds a section header's size in bytes. A section header is one entry in the section header table; all entries are the same size.
- **e_shnum**: This holds the number of entries in the section header table. Thus, the product of e_shentsize and e_shnum gives the section header table's size in bytes.
- **e_shstrndx**: This holds the section header table index of the entry associated with the section name string table.

2.2.2.2 ELF Program Header

The ELF program header, as shown in Table 2.7 tells the Linux binary loader where to store the program in memory along with setting any specific permission(s) for that segment of memory.

- **p_type**: This tells what kind of segment this array element describes or how to interpret the array element's information.

1										2																				3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
p_type																															
p_offset																															
p_vaddr																															
p_paddr																															
p_filesz																															
p_memsz																															
p_flags																															
p_align																															

Table 2.7: ELF Program Header

- **p_offset**: This gives the offset from the beginning of the file at which the first byte of the segment resides.
- **p_vaddr**: This gives the virtual address at which the first byte of the segment resides in memory.
- **p_paddr**: On systems for which physical addressing is relevant, this is reserved for the segment's physical address.
- **p_filesz**: This gives the number of bytes in the file image of the segment; it may be zero.
- **p_memsz**: This gives the number of bytes in the memory image of the segment; it may be zero.
- **p_flags**: This gives flags relevant to the segment.
- **p_align**: Loadable process segments must have congruent values for p_vaddr and p_offset modulo the page size. This member gives the value to which the segments are aligned in memory and in the file.

2.2.2.3 ELF Section Header

The layout of the ELF binary format is similar to the Windows PE format but there is an important difference. In the Windows PE format, we mentioned that a

1										2										3										3	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
sh_name																															
sh_type																															
sh_flags																															
sh_addr																															
sh_offset																															
sh_size																															
sh_link																															
sh_info																															

Table 2.8: ELF Section Header

section can be placed inside another section. This was allowed by the Table pointers in the Optional Header. With regard to ELF, there is no such list of tables pointing to places in the binary file for sections that must be loaded into memory. Any segment of information that must be handled by the Linux binary loader in a particular way needs to have its own section header. The ELF section header is shown in Table 2.8.

In the Windows PE format, the import table can be hidden by a utility preparing a self-modifying program inside another section. This type of subterfuge is not possible in the ELF format. Any shared libraries that need to be loaded must be described in the ELF section header called '.dynamic'.

- **sh_name**: This specifies the name of the section. Its value is an index into the section header string table section giving the location of a null-terminated string.
- **sh_type**: This categorizes the section's contents and semantics.
- **sh_flags**: Each section supports 1-bit flags that describe miscellaneous attributes.
- **sh_addr**: If the section will appear in the memory image of a process, this gives the address at which the section's first byte should reside. Otherwise, the contains 0.
- **sh_offset**: This value gives the byte offset from the beginning of the file to the first byte in the section.

1										2										3									
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1
magic																															
minor_version																major_version															
constant_pool_count																constant pool item #1															
...																constant pool item #N															
access_flags																this_class															
super_class																interfaces_count															
interface #1																...															
interface #N																field_count															
field_info #1																...															
field_info #N																methods_count															
method_info #1																...															
method_info #N																attributes_count															
attribute_info #1																...															
attribute_info #N																															

Table 2.9: Java Class File format

- **sh_size**: This gives the section's size in bytes. Unless the section type is SHT_NOBITS, the section occupies sh_size bytes in the file. A section of type SHT_NOBITS may have a non-zero size, but it occupies no space in the file.

- **sh_link**: This holds a section header table index link, whose interpretation depends on the section type.

- **sh_info**: This holds extra information, whose interpretation depends on the section type.

2.2.3 Java Class

A Java Class file, as shown in Table 2.9, is different from the other two formats, Linux ELF and Windows PE formats, in that every piece of information is contained in one structure. There is no concept of sections, segments, or any data structure other than a Class file data structure.

2.2.3.1 Class Header

- **magic**: The magic item supplies the magic number identifying the class file format; it has the value 0xCAFEBABE.
- **minor_version**: Minor version of the class file.
- **major_version**: Major version of the class file
- **constant_pool_count**: The value is equal to the number of entries in the constant_pool table plus one.
- **constant_pool**: This is a table of structures representing various string constants, class and interface names, field names, and other constants that are referred to within the Class file structure and its substructures.
- **access_flags**: The value is a mask of flags used to denote access permissions to and properties of this class or interface.
- **this_class**: The value of this item must be a valid index into the constant_pool table. The object at this index must be a Constant Class Info type.
- **super_class**: For a class, the value of the super_class item either must be zero or must be a valid index in the constant_pool table. For an interface, the value of the super_class item must always be a valid index in the constant_pool table.
- **interfaces_count**: Number of interfaces in the class interfaces. Each value in the interfaces array must be a valid index into the constant_pool table. Each object in the constant_pool must be of Constant Class_Info.
- **field_count**: Number of fields in the class
- **methods_count**: Number of methods in the class
- **attributes_count**: Number of attributes in the class

The "major_version" and "minor_version" are used by the Java Virtual Machine (JVM) to determine if it is capable of running the java byte code. The Java Class file might have a higher version number than it is capable of executing properly. The JVM starts reading the constant_pool, similar to a table of contents, for validating, initializing and preparing the class file for execution.

2.2.3.1.1 Constant_Pool: The constant_pool is a list of all the classes, interfaces, fields, methods and data that can be used in the Java Class file. Each item in the constant_pool is a variable length constant_pool_Info object. Each object has an 8-bit tag describing the kind of constant_pool item it is along with a series of bytes giving the information for that object. The Java Virtual Machine (JVM) reads in all this information into a run time constant_pool for use later.

It is interesting to note that the Java Class file explicit separates instructions of a program from its data. Each method_info structure contains a list instructions stored in the Code_Attribute. This attribute is only allowed to contain valid 8-bit Java byte code instructions. No data is allowed to be stored in this attribute.

A method can reference constant values found in the constant_pool for use in the function. Any data used by the method will be found either in the constant_pool, in the stack frame or on the heap.

The validation process of the JVM that validates all the data structures in the Java Class file follows a strict set of rules. The discussion of the rules is beyond the scope of this dissertation.

2.3 Compressed Executables

Malicious code present in an executable has the capacity to oppose the security aims of confidentially (by revealing information against the wishes of the owner), integrity (by disrupting or altering data), and availability (by rendering systems unusable) [4]. Therefore, it is both natural and imperative to seek to root out and defend against such attacks.

There are a variety of methods that have been proposed and applied (with varying levels of success) in order to improve the classification of and defense against malware. The general components of defense systems, as presented in [43] include:

1. Data collectors/information sources which provide information on normal and abnormal entities.

2. Analysis components which look at new instances and attempt to determine whether or not they are a threat.

3. Response components which take action against wouldbe threats

4. Testing components which attempt to expose vulnerabilities in the system.

Data collectors typically provide information to the analysis components which in turn provide detection capabilities to the response components. The response components then take appropriate action against a given threat. Finally, the testing components identify weaknesses in the system and provide information to the data collector and analysis components.

One focus of this work is to improve analysis capabilities by providing information about executable files, even though little information concerning those files may be known. While techniques exist for disassembly and reverse compilation for executables, different tactics must be taken for self-modifying files. If a reverse engineering framework has the capacity to know exactly the type of file with which it is dealing with then the process of reverse compilation becomes simpler. The framework can take action using the discovered knowledge of the file to recover the hidden instructions. Otherwise the framework has to take a general approach which may or may not work. Reverse engineering a file supports digital forensic investigation since it allows an analyst to determine the intention of a malicious file without actually executing the file. Additionally, knowing that an executable has been compressed can alert an investigator of suspicious activity on a machine under investigation. Normally most executables

are stored in a uncompressed format on a system since space is not an issue. Embedded system designer may choose to compress their application since the target device often has limited storage space. Therefore when an investigator finds an compressed executable on a system that does not normally need them it should alert them to a potential problem.

Static analysis of files typically focuses on determining program properties and behavior by examining software without running it, while dynamic analysis executes a program repeatedly under known conditions and then observes the results [24]. There are advantages to both approaches. In static analysis, program execution must be abstracted and modeled, so the results may be less specific in order to preserve correctness. Put simply, it may be difficult to determine exactly what a program does in all situations without actually running it. There are other complications, as illustrated in [66]:

- Tractability: increasing program size dramatically increases the time required for static analysis.
- Decidability: it is impossible to know how many times a loop will be executed at run time; it is equally impossible to know how many times a function may be called recursively.
- Pointer Aliasing: in languages that are not type-safe (C, for example) it is possible to have many different variables accessing some buffer in memory, and it is infeasible to gain an accurate listing of such variables.

In dynamic analysis, while it is typically easier to get a clear picture of what the program does, given specific test cases, it is generally unreasonable to expect to be able to exhaustively test all possible executions. Worse still, in the domain on information security, it is highly undesirable to execute potentially malicious software many times in order to examine its behavior. Thus, we will largely confine our discussion of related work to static analysis and reverse engineering schemes.

2.4 Survey of Decompiling Architectures

This section covers the different kinds of decompiling architectures. There are two types of decompiling architectures covered in reverse engineering research. Section 2.4.1 covers the decompiling architectures for x86 problems and Section 2.4.2.

2.4.1 x86

Sung and Mukkamala [55] demonstrates the complexity of reverse engineering code in general. It is highly desirable from a security standpoint to be able to have access to the source code of a potentially dangerous executable.

Some decompiling architectures focuses on binaries that are compiled for the Intel x86 architecture. The following research listed below investigated methods to reverse engineer x86 binary programs into a form of high-level output (e.g. C programming language).

2.4.1.1 Cifuentes

Summary: Cifuentes [14] illustrates one example of x86 decompiling architecture to transform from x86 executable code to high-level C language code. The stepwise process converts the application to assembly first. Secondly it converts the assembly output into RTL form in a control flow graph. Finally it converts the RTL in the control flow graph into a high level form of RTL. The output is converted into C programming language.

Cifuentes's work [14] demonstrates the complexity of reverse engineering code in general. It is highly desirable from a security standpoint to be able to have access to the source code of a potentially dangerous executable. However, while creating executables from source code is easily done with modern software, the reverse is not at all simple. Cifuentes illustrates a working process to move from executable code to high-level language code.

Many of the steps are simple in theory but difficult in practice due to a variety of issues. Converting from binary to assembly can be difficult, since it is difficult to distinguish between data and instructions in the binary format. System-specific strategies make this possible but still difficult. Compression and encryption techniques make this effort even more difficult, especially if the compression or encryption method used on the file is not known (thus the motivation for this research). A further example of the complexity of this process comes in converting from assembly to high-level code. This is often made difficult by the fact that not all assembly instructions have an equivalent high-level representation. Further, differing choices of high-level language will lead to differing final representations.

Even given these difficulties, Cifuentes manages to successfully use a variety of new and existing tools to effectively reverse engineer a number of programs of varying complexity. However, as is expected, there are issues with scalability and such efforts have the potential to be somewhat fragile with respect to which executables can be reverse engineered.

Key Contributions: A three stage process for automatically converting DOS x86 executables into C programming language:

- The first stage is the "Input" where the binary program is converted into a control flow graph containing instructions in RTL format.
- The second stage is the "Analysis" where the control and data flow analysis transform the control flow graph.
- The final stage is the "Output" where the contents of the control flow graph are converted into a file containing C programming language instructions.

Weaknesses: Due to a variety of issues some of the above steps are simple in theory but difficult in practice. Converting from binary to assembly can be difficult since it is difficult to distinguish between data and instructions in the binary format. System-specific strategies make this possible but still difficult. Compression and encryption techniques make this effort even more difficult, especially if the compression or

encryption method used on the file is not known. A further example of the complexity of this process comes in converting from assembly to high-level code. This is often made difficult by the fact that not all assembly instructions have an equivalent high-level representation. Further, differing choices of high-level language will lead to differing final representations. Even given these difficulties, Cifuentes manages to successfully use a variety of new and existing tools to effectively reverse engineer a number of programs of varying complexity. However, as is expected, there are issues with scalability, and such efforts have the potential to be somewhat fragile with respect to what executable programs can be reverse engineered.

2.4.1.2 Boomerang

Summary: Boomerang [35] is a program which attempts to extend the dissertation work of Cifuentes. It is designed to work with either Intel Pentium or Sparc binary executable programs.

Key Contributions: Boomerang follows the same structure as Cifuentes's work:

- It first converts an x86 (Linux/Windows), PowerPC (Linux/Mac), or Sparc (Solaris) executable to assembly using a disassembler.
- Secondly it translates the assembly into an intermediary language for the purpose of reducing the bulk of instructions by analyzing the semantics of the operations.
- Next it converts the intermediary language into arbitrary high-level code using a decompiler.
- Finally it post-processed arbitrary high-level code into C language for analysis and use.

This program was utilized to recover the core algorithm from a third-party's binary application along with almost all original class names and the complete class hierarchy. [21]

Weaknesses: It suffers from the same issue of scalability of Cifuentes's work. It cannot be altered to add new functionality to apply this solution to other problems without serious modification. It does not support compressed or encrypted executables.

Andromeda Decompiler

Summary: The goal of the Andromeda Decompiler [51] is to create a universal interactive decompiling environment. The idea is to take in a user's binary executable and produce output in a high-level language of their choosing. The program runs on the Windows platform and supports 32-bit Intel x86-compatible programs and C/C++ output.

Key Contributions: The two main features of Andromeda Decompiler are:

- Research and investigation of binary modules at a level of source codes.
- Partial or full restoration up to recompilable forms.

Weaknesses: All the features of the system were gleaned from the website for the project. There is no source code available for the project so the internal workings of the decompiler could not be investigated. At the time of this dissertation, no work had been done on this project in four years given the last release was in 2005.

Static Detection of Malicious Code in Executable Programs

Summary: The static analyzer developed by Bergeron, Debbabi, Desharnais, Erhioui, Lavioe and Tawbi [3] converts a Windows PE executable file into an intermediate representation, creates a control flow graph from this information and analyzes the graph against a policy to determine if malicious code is present.

Key Contributions: The authors provide:

- A three step process for converting a Windows PE into a control flow graph for analysis using the IDA32 disassembler.

- The first step processes a syntax tree representing the control structure of the program. Each basic block in the control structure contains assembly instructions produced by the IDA32 disassembler.
- The second step performs the control flow and data flow analysis on the control flow graph.
- The final step of the process is to apply a security policy to the control flow graph to detect malicious code.

Weaknesses: The solution is exclusively catered to x86 executable binaries in the Windows PE format. Its design is very similar to Cifuentes [14] altering only the method of producing the x86 assembly instructions. Whether or not the analyzer can be expanded to Linux and Java is not clear.

Analysis of Binary Programs

Summary: Kapil and Kumar [32] produced a decompiling system composed of an disassembler library *libdisasm* and a translation phase using the Standford University Intermediate Format (SUIF). The system converts a binary program to an Intermediate Representation(IR).

Key Contributions: The SUIF is a:

- Modular and extensible system that can describe a wide set of CPU instructions.
- The system utilizes these two tools in a two pass structure that produces the assembly output of the binary program which is processed as a control flow graph.
- The output of the program is in SUIF format for conversion later into a higher level language.

Weaknesses: The authors state that the system cannot handle input programs that contain pointers, recognize user defined structures, recognize defined return type for functions, handle arguments to functions passed in memory, identify loops that contain *break* or *continue* statements, and handle control flags changed by all statements and

recognize if-then-else block structures. It only assumes control statements alter control flags.

Designing an object-oriented decompiler

Summary: Desquirr [23] produced an IDA Pro plugin that performed a series of transformations and analysis to produce high level output.

Key Contributions:

- Conversion method for translating assembly instructions into code in a high-level language. It first translates 16- and 32-bit Intel 386 machine code produced by the IDA Pro disassembler into Register Transfer Language (RTL) form.
- Secondly, it performs data and control flow analysis before finally producing source code written in the C programming language.

Weaknesses: Desquirr cites Cifuentes work as a motivator for creating his solution but limits his work to using IDA Pro to reverse engineer x86 programs. Since IDA Pro provides a set of binary file formats and disassembly of the binary input, Desquirr's program utilized this through the IDA plugin interface. Instead of starting the reverse engineering from the first byte of the code section as done by the Linux program objdump, the program starts reading from the entry point of the target application. Desquirr uses the same analysis operations that Cifuentes uses in her tool. These tools separate code into basic blocks, perform live register analysis, find Def-Use chains, register copy propagation, find functions and finally produce assembly instructions broken up into individual functions. This solution does not support analysis of other architectures (e.g. PowerPC) or other file formats (e.g. Linux ELF).

Cobra: Fine-grained Malware Analysis using Stealth Localized executions

Summary: Cobra is a dynamic fine-grained malicious code analysis framework that executes in kernel mode. The goal of the project is to support multi-threading,

self-modifying / self-checking code and any form of code obfuscation in both user and kernel mode on commodity operating systems [64].

Key Contributions:

- Method for breaking up code execution into blocks for analysis.
- Support of self-modifying code through the use of xfer-stubs which allow Cobra to always have control at the end of a block's execution.
- Selective execution of code based on user defined points.
- Call back feature for applications built on its API.

Weaknesses: While the method of analysis for malicious programs is an excellent solution, it is not scalable. The problem lies in the possible combinations of architectures and operating systems. The Microsoft Windows family of operating systems is dedicated mainly for the x86 architecture for home and business computers. The Windows operating systems also operate in the embedded products. In order to support the analysis of malicious code in all possible combinations, Cobra would have to be ported to each of the possible operating systems. It also relies on the host operating system to prepare the malicious program for execution and actually executes each of the instructions. While the strength of dynamic analysis of malicious programs is more powerful than static analysis, there is an inherent risk in actually executing the instructions on the host operating system even if it was in a VM environment. The risk of executing malicious programs is that it is not known which, if any, of the host and network services will be used.

Detecting Malicious Code by Model Checking

Summary: Model checking can be used to provide a flexible way to detect malicious code patterns in executables. [33]

Key Contributions: The authors contribute:

- Computation Tree Predicate Logic (CTPL) to extend the Computation Tree Logic (CTL) [16]. It is as expressive as CTL but allows a precise way to represent malicious code patterns.

Weaknesses: CTPL provides a way to represent malicious code in a precise way describing what a program should not be doing. To fully exercise this method, all possible "malicious" patterns would need to be determined. In addition, the possibilities of no false positives or negatives would have to be assured. The authors do not provide any tests where normal programs, not containing malicious code, were processed by the system. There is no proof given that the system will not accidentally classify something as malicious code nor incorrectly classify a malicious program as a virus free. It is uncertain that this solution would be capable of being used for Java binaries which use different types of assembly operations.

2.4.2 Java Class

Java program files are platform independent files compiled for Java opcodes to run on a Java Virtual Machine (JVM). Decompiling programs convert the Java opcodes either to Java assembly or back into Java source files. The follow sections describe various solutions to decompiling Java Class files into source files.

Sun Disassembler (Javap)

Summary: The Javap program is a part of the Sun Microsystem JDK that will disassemble the input class file to the Java assembly. Since it is packaged within the JDK, it is safe to assume that the expected input is one that was compiled with the JDK compiler (javac).

Key Contributions: The Javap programs contributes:

- A Java disassembler which converts a Java class file into a Java source file containing Java assembly instructions.

Weaknesses: Javap does not produce high-level Java source code which can rapidly aid understanding of a suspect Java Class file. Javap produces a output file containing java assembly instructions from a given Java Class file. An assembly instructions are one step up from the 1s and 0s a CPU or Virtual Machine instruction. It is not very easy to identify high level programming constructs when looking at the assembly instructions. For example identifying a complex if statement.

Dava decompiler

Summary: The Dava [41] decompiler project is a multiple stage decompiling process by first constructing a typed list of all the statements found in the Java Class file. Secondly, it constructs a control flow graph from the the output of the first stage. Finally, it performs the analysis on the input to produce their high level output.

Key Contributions: The authors provide:

- data structure called Structure Encapsulation Tree (SET) that contains the Java structures as the file is reverse engineered.
- six stages of transformation
    1. Synchronized Blocks which contain the regions of the instructions of a synchronized() block.
    2. Loop Detection (e.g. while)
    3. Remaining Conditionals identification (e.g. if)
    4. Exceptions
    5. Statement Sequences
    6. Continues, Breaks, and Arbitrary Labeled Breaks

Weaknesses: The authors dive straight into the decompiling. They do not cover what compiler was used. The basic problem with their work is that it is not obvious how these same techniques will apply to other executable binaries compiled for different architectures (e.g. x86 program for Windows PE).

In the course of explaining their project, the authors talk about what has been previously done in the area of reverse engineering architectures. They talk about Cifuentes's work as an example of restructuring an input binary to source code based on gotos. Cifuentes was targeting C as an output programming language hence the use of goto statements to handle loops in the program. The Dava authors mention that Cifuentes's approach would require modification to handle Java. This is true because C binary executable and Java Class files are vastly different. Cifunetes's input phase could not handle the Java Class file. It would require a new front-end to produce a new control flow graph for the rest of her architecture. The same argument can be applied to Dava because it cannot handle C binary executable programs. Each group of researchers has produced a customized solution that cannot be easily altered. So the Dava authors have no space to argue that they are doing more. Both groups scoped their problem in order to make a tractable solution.

The fast JAva Decompiler (JAD)

Summary: JAD [34] is a Java decompiler written in C++ that is used to produce source code or Java assembly output from a valid Java Class file.

Key Contributions:

- Enhanced readability of the generated source code.
- Ability to comment Java source code with JVM byte codes. Useful for verification and educational purposes.
- Full support for inner and anonymous classes.
- Fast decompilation and simple setup.
- Automatic conversion of identifiers garbled by Java obfuscators into valid ones.

Weaknesses: While JAD can decompile Java Class files into Java source code, it has a list of limitations. It cannot handle zip or jar files as input to the system nor make use of the Java Class hierarchy. It has trouble with nested classes that

contain constructors with extra arguments added by the Java compiler. Also it makes no mention about determining the compiler used to create the input class file. JAD ignores line number table and source file attributes in the class file. Finally JAD has trouble with in-lined functions.

DJ Java Decompiler

Summary: The DJ Java Decompiler [45] is a Java decompiler and disassembler. It will recreate a file containing equivalent source code from a class file. It is a front-end application that utilizes the JAD Decompiler as its decompiling engine. It makes no mention about determining the compiler used to create the input class file.

Key Contributions: The DJ Dava Decompiler is a:

- Stand alone Windows program able to decompile Java Class files without requiring the Java JDK to be installed.
- Provides an editor to allow changing the generated Java code.

Weaknesses: The design is only able to handle Java Class files and not any other executable binaries.

Reverse Engineering Compiler (REC)

Summary: REC [10] is a graphical decompiler that can handle Intel x86, 68k, PowerPC and MIPS R3000 programs for ELF, COFF, PE, AOUT, Playstation PS-X and raw binary data files.

Key Contributions: REC is a:

- Graphical decompiler that can handle Intel x86, 68k, PowerPC and MIPS R3000 programs for ELF, COFF, PE, AOUT, Playstation PS-X and raw binary data files.

Weaknesses Since the source files for REC are not publicly available, there is no way to understand the internal behavior of REC. REC is designed to best handle

programs that were originally written in the C programming language and compiled with the debug options enabled. If a target executable binary does not have debugging information present, it cannot produce good output.

Decompiler for TurboC (DisC)

Summary: DiSC [36] is a decompiler specialized for the TurboC compiler.

Key Contributions:

- Only handles MS-DOS programs originally written in the C programming language and compiled with the TurboC compiler.
- DiSC is a prime example of a decompiler written specifically for a certain compiler. DisC will produce TurboC output that only requires a few modifications by the user to compile the resulting

  source code.

Weaknesses: It does not recognize floating point code and strings. It is not 100% automated and is limited to TurboC. Most decompilers focus on convert instructions for a particular processor into assembly language. They are not typically customized to decompile an application produced by a particular compiler.

Java Optimize and Decompile Environment (JODE)

Summary: JODE [29] is an optimizer for Java Class files to remove debug information, renaming objects like classes or variables, and removing dead code.

Key Contributions:

- Can be used to decompile Java Class files
- Can optimize Java Class files.

Weaknesses: It is limited to Java Class files that do not contain Java generics.

Mocha

Summary: Mocha [63] is a Java Class decompiler originally written by Hanpeter van Vliet.

Key Contributions: Mocha contributes

- Mechanism to decompile Java Class files into Java source code.

Weaknesses: It is limited to Java Class files that do not contain Java generics. The solution was created before Java generics were a part of the Java programming language. The source code for this project cannot be obtained due to the author's death in 1996. When JAD was applied to the Mocha's Java Class files, the resulting code was determined to be obfuscated. Any further analysis into how Mocha works would require a lengthy reverse engineering process.

2.4.3 Static Slicing on Binary Executables

Summary: In [4], the authors propose a process for static slicing of binary executables in order to detect malicious code very similar to the approach offered in [14]. The difference is that in [14], the author was seeking the source code as the end product, whereas in [4] the authors are interested in the identification of security vulnerabilities in software. However, this approach is intended for use on commercial off-the-shelf software, where the source code is unlikely to be available directly for use in analysis.

Key Contributions: Their methodology is as follows:

1. Using disassembler software to reduce binary code into an assembly equivalent.
2. Transforming the assembly code into a corresponding high-level representation, which maintains the semantics of the code while improving the quality of the analysis.
3. Using the high-level representation, they attempt to identify potentially malicious portions of the code. This serves to reduce the amount of code which must be further analyzed (thus reducing the complexity of the overall file analysis).

4. Examining potentially malicious slices identified before.

Weaknesses: While this approach does provide a methodology for dealing with software when the source code is not available, it does have certain weaknesses. First, it is highly dependent on the availability and capability of tools, such as disassemblers and assembly-to-source translators to be able to successfully perform analysis. In practice, these tools may not always be available, often have difficulties with scalability, and tend to be very system-specific. Further, these strategies may not be effective at all against purposefully altered executables (such as those that have been compressed, obfuscated, or encrypted), since they depend on disassembly.

2.4.4 Static Analysis on Executables without Decompiling

Analyzing an executable file directly without decompilation improves the robustness of the solution and can provide greater insight than static analysis on the source code.

2.4.4.1 Related Work on Windows Portable Executable Files

Several of the other approaches surveyed depend either on the availability of source code directly or on the ability to reverse-compile such source code. However, as shown by [14], this is a non-trivial effort. Further, in a security environment, this process may not be at all practical for file analysis. To this end, the authors in [59, 60] attempt to extract useful security information directly from Windows PE executables without decompilation or execution.

The motivation behind this work is that even when source code is obtainable, it may not always reveal the vulnerabilities in the executable. For example, malicious behaviors may be inserted in the executable post-compilation (many viruses take this approach). Tevis automatically extracts information from the PE file to produce a file synopsis. Then, anomaly detection is done by a series of discovered rules:

- Checking for mismatches in table sizes as reported by the optional headers versus the actual size in memory.

- Checking for large blocks of memory unaccounted for in the header (which may potentially be used for holding data in a self-modifying file).
- Checking for sections in the Windows Portable Executable sections header which are both writable and executable (also potentially used for self-modification).
- Analyzing the symbol table for strings of potentially dangerous functions (for example, notoriously unsafe C functions like gets()).
- Identifying purposefully removed information in symbol tables and import tables, which may indicate an attempt to hide inappropriately placed data.

Tevis tested a test-set of thousands of Windows Portable Executable (PE) (discussed at greater length in Section 2.2.1) files for anomalies and found no shortage of vulnerabilities in a large array of even Microsoft system files. His work is particularly of interest since he demonstrates that a large amount of information can be extracted and interpreted from the PE header file itself. This shows that patterns and anomalies can be observed directly from the executable. The downside of this research is that the work does not naturally translate well to other executable types or other operating systems. All trends and rules would have to be rediscovered for any other types. Another problem, in terms of deployment, is that while the system attempts to identify anomalies, it does not attempt to draw many conclusions about those anomalies. Without further analysis, these anomalies would produce a high false-positive rate. They would constantly alert a user of potential problems with executables, whether or not an immediate security concern actually existed.

2.4.4.2 Related Work on Self-Modifying Virus Detection

Static analysis has been proven to work in the detection of self-encrypting and polymorphic viruses by the work presented in [42]. A self-modifying virus has its payload encrypted and carries code for decryption once it is in memory. This means that most virus detection utilities will be ineffective as they cannot perform code checking in

an attempt to determine the intent of the program. Polymorphic viruses are enhanced self-modifying viruses that were designed to avoid fixed patterns which are often used by virus scanners/detectors. The authors of this work created a code simulation environment in order to analyze the actions performed by a number of programs. The authors used a simulator that was allowed to run virus code in a controlled environment in order to detect external API function calls. Also the authors allowed for execution of these API function calls by OS emulation that uses stub functions instead of the real library functions.

The results presented in [42] show that it is possible to use static analysis in order to find viruses that are self-encrypting and/or polymorphic. In a testing set where a majority of the instances (80%) had never been seen by the tool and using only a handful of the policies that were defined, the system was able to detect malicious code with over 80% accuracy and as high as 95% accuracy with four of the policies defined.

2.5 Machine Learning Related Work

In [58], the authors describe the use of a Neural Network to analyze the boot sector of a PC to determine if it was infected by a virus. In this application, their representation of a training instance was novel, since simply including all the 512 bytes of the boot sector would be infeasible due to Bellman's Curse of Dimensionality [2]. Instead, they interpreted the boot sector as a series of approximately a dozen features (properties of the boot sector) and classified based on this set. Their primary focus was the minimization of false positives, since they were targeting a commercial audience. They used a training set of 200 known boot-sector viruses and roughly 100 executable viruses. The neural network used was trained with an iterative back propagation algorithm. This system resulted in a very low false-positive rate (less than 1%) and a modest false-negative rate (around 15%).

In [50], the authors attempt to identify malicious executables using a strategy of feature extraction from a large test set of binary executables. Training instance features include Dynamically Linked Libraries (DLL) file names used, DLL function calls made,

numbers of function calls made to specific DLLs, and byte sequences appearing in the files. These test sets were used to train both Naíve Bayesian Classifiers and Multi-Naíve Bayesian Classifiers (probabilistic instance-based learners), which ultimately outperformed signature based malware-detection programs dramatically in terms of detection rates (although they demonstrate slight false-positive rates).

In [55], the authors attempt to reduce the number of elements in training instances (to be used for a Neural Network classifier) by assessing their importance. If smaller training instances can be produced without sacrificing accuracy, then this practice can improve the computational speed of the classifier when being used for classification (of course, this will improve the speed of training generally as well). Better still, by removing useless data from the learner, it can actually improve accuracy. In order to rank features, they use a feature selection scheme designed to maximize classification ability. This research is significant since it examines an effective mechanism for reducing the size of training instances (they generally saw as large as 20% reductions in the number of features without loss of classification accuracy).

2.6 Survey of Compiler Detection

A thorough search of unclassified sources and published literature was conducted as demonstrated in this chapter. Queries were sent to the open source community and Richard Stallman [54] on the belief that someone had conducted this work. The answers received back showed this was not the case.

2.7 Generic Decompilation

Construction of a general decompiling / reverse engineering architecture that can be applied across a wide set of inputs and various problems has not been done. Similarly, detection of which compiler was used to create a binary executable has not been done. Clearly the previous work in this area shows that reverse engineering is possible to

some degree. This work will show how previous reverse engineering solutions can be generalized.

2.8 Summary

Past research into reverse engineering have shown how specific solutions can solve certain reverse engineering problems. The solutions were catered to reverse engineering files written in a specific file format. Each file format provides a lot of information that can be utilized in a generic reverse engineering architecture to verify certain properties about executable files. Machine learning can take advantage of this information to classify and verify properites found in executable files.

Chapter 3

Analysis And Classification

3.1 Introduction

In this chapter, we describe the machine learning technique utilized, the inner workings of a General Regression Neural Network, and optimization techniques used to optimize the parameters for the GRNN.

3.2 The General Regression Neural Network

Neural networks, in general, attempt to emulate the biological neural networks in the brain, which depend on massive parallelism and inter-connectivity in order to process information quickly.

A GRNN, shown in Figure 3.1, is a one-pass learning algorithm which attempts to approximate a continuous variable that is dependent on many representative vector training instances. This is more beneficial than other neural network strategies which require iterative learning that uses many iterations in order to produce workable solutions. In practice, any analysis system deployed to provide computer security should be created with the awareness that computational resources are not infinite. Typical computer users will not sacrifice much convenience for security.

As described by Specht[53], elements of training instances $x_1 ... x_n$ are passed into the network and collected into pattern units (where each unit represents a single training instance). The output from the pattern units is collected in the summation units (as described below) and then the output of the summation units is ultimately collected to provide an estimate for $\hat{Y}$ corresponding to an unseen input vector X.

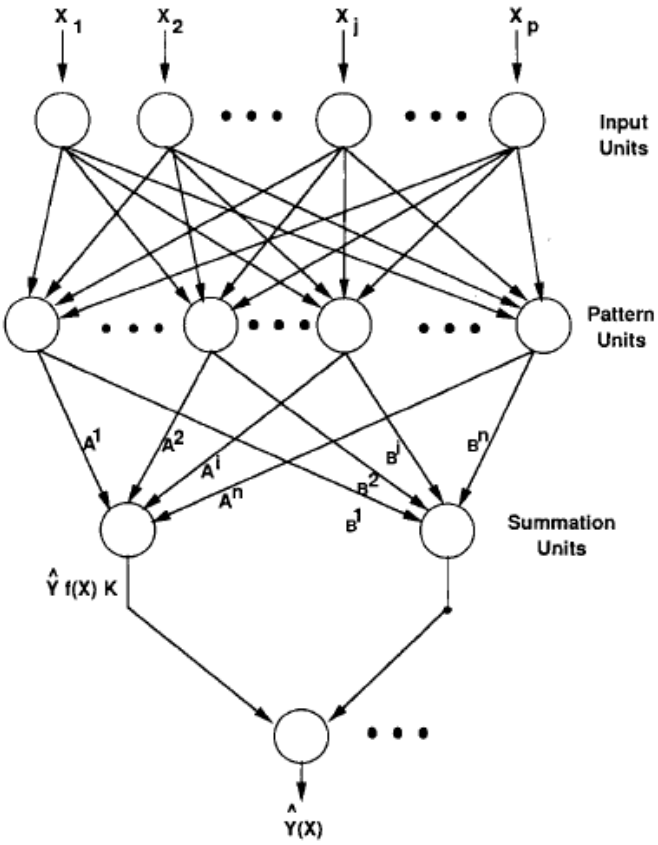


Figure 3.1: A GRNN Block Diagram

Each training instance in the GRNN consists of a vector X and a desired output Y. In order to provide an estimate of the value of $\hat{Y}$ for an unseen instance X is calculated in Formula 3.1:

$$\hat{Y}(X) = \frac{\sum_{i=1}^{n} Y_i * exp(-\frac{C_i}{\sigma})}{\sum_{i=1}^{n} exp(-\frac{C_i}{\sigma})} \tag{3.1}$$

where n is the number of all training instances, Y_i is the desired output for a given training instance vector, σ is a constant parameter of the GRNN, and the value of C_i is the Euclidean distance between X and a given instance vector, as shown in Formula 3.2:

$$C_i = \sum_{i=1}^{p} |X_j - X_{ij}| \tag{3.2}$$

where X_j represents a single element of the training instance vector and X_{ij} represents the corresponding element in the instance to be classified.

The effectiveness of a Neural Network depends on [28]:

1. The presence of learnable characteristics in the training instances
2. The number of training set instances
3. The representation of the training set
4. The complexity of the problem at hand
5. System parameters of the Neural Network

Our classifier attempts to carefully address each of the above factors.

3.2.1 Learnability in the Training Instances

One of the significant risks in the initial phase of this research was that the information available from the Java Class, Linux ELF and Windows PE file headers would not be sufficient to provide the learning algorithm with enough knowledge to effectively classify target (e.g. compiler or compression method). The algorithm used depends on

discovering relations between various fields in the file headers. If there is not enough information from the file headers that a learner can form these relations then this method would fail. Even the best learner cannot extract information if there is not a correlation between the input vectors and the desired outputs.

In regard to compressed file classification, there was reason to believe that there is a correlation between the physical and virtual sizes of files in the executable (to determine between compressed and uncompressed) and that this information could be effectively extracted. Perhaps of greater interest was the hope that each compression algorithm had a sufficient characteristic impact on the PE Header to be revealed to the neural network.

In a similar light, there was reason to believe that each compiler leaves unique characteristics in the file headers that can be revealed to the neural network. Each compiler has a unique way it writes a binary computer application file for a particular file format given the same input source code file. This is because groups of programmers have different methods for solving the same problem. Given the same plans people often interpret them differently. Therefore, it was likely that there would be unique characteristics found in the file headers that could be used to classify the compiler.

3.2.2 Training Set Instances

Under ideal conditions, the training set should be relatively large and perfectly representative of all possible instances that the network could be asked to classify. In practice, this is not always possible. We sought to choose a wide variety of executable files for the two problem areas. In the compressed executable classification we sought both compressed and uncompressed of common applications on a Windows XP system for the compression detection. In the compiler classification we sought to choose a wide variety of Java Class (produced from Java source files), Windows PE and Linux ELF executable files (produced from C++ source files).

3.2.3 Complexity

In general, the complexity of the classification problem is not under the control of the system designer. In the first problem, the system attempts to differentiate between uncompressed files and compressed files that were compressed using one of three common utilities. Thus, the system must attempt to classify a given executable into one of four equivalence classes. In the second problem, the system attempts to differentiate between the four Java compilers if given a Java class file, three C++ compilers that produced a Windows PE file and three C++ compilers that produced a Linux ELF file.

3.2.4 Network Parameters

The General Regression Neural Network is dependent on the value chosen for σ. σ is a system parameter of the GRNN which impacts performance in terms of success rate. Larger values of σ essentially mean that more nearby training instances (in the sense of Euclidean Distance) will contribute to the resultant output. Essentially, large σ values act as a smoothing function, which can be desirable when the available training data is either sparse or contains outliers. When σ is small, it reduces the contribution of nearby instances to virtually zero; therefore, only the closest instances will provide any contribution at all. When the training set is highly representative of the classification function being considered, a small σ would be highly desirable. For all cases other than the above, some intermediate value of σ would be appropriate. No prior information regarding the appropriate value of σ was known in this case, so optimization methods were utilized to find a good setting. These approaches are described in Section 3.3.

3.3 The Evolutionary Hill-Climber Algorithm

Although there are many techniques for parameter optimization, and although only one parameter was being optimized (σ), a simple Evolutionary Hill-Climber (EHC) was utilized. For comparison, a logarithmic search approach which searches a larger portion of the search space was also implemented and compared. The rationale behind using

the EHC was in recognition that this framework might need to be extended to deal with a more complex search. For example, for other similar problems, it might be interesting to determine not only network parameters, but which attributes are useful for the classification problem itself (such as exactly which fields in the PE Header should be used). In this case, a simple logarithmic search may not be able to effectively find the best solutions (particularly as the number of possible attributes becomes quite large).

3.3.1 Algorithm Description

Traditional deterministic Hill-Climbing methodologies tend to inadequately deal with complex search spaces and instead get stuck at local minima [27]. Instead, stochastic methods of search often prove preferable in instances where there exists no guarantee of smoothness or of singular maxima in the function to be optimized [48]. In optimizing σ for the GRNN, there are no such guarantees. The Evolutionary Hill-Climber (EHC), based loosely on the Random-Mutation Hill-Climber described in [26], first randomly creates a single candidate solution consisting of a chromosome with a number of values corresponding to parameters being optimized (in this case, only the σ value). Initial values are chosen from within the allowable range dictated by the problem type (in this case, the range of values for the σ was chosen from the range (0,1.0). This candidate solution is then evaluated by a fitness function, and the fitness is assigned to that individual. This is shown in Algorithm 1.

Algorithm 1 Evolutionary Hill-Climber

```
1:  procedure EHC
2:      t = 0
3:      Initialize candidate solution i
4:      Evaluate i
5:      while GRNN evaluations remain do
6:          choose a step location
7:          evaluate step location
8:          if step yields better fitness
9:          replace i with new individual
10:     end while
11: end procedure
```

On each iteration, a single uniform mutation (taken from the range [0,1]) is added to the parameter value, multiplied by the mutation amount δ and the starting value of the gene itself. The value of 0.25 for δ was chosen from the set [0.03,0.05,0.1,0.15,0.25,0.5] by experimentation. This new candidate solution is then evaluated. If the new candidate solution has better fitness than the previous, the previous is replaced. However, if the new candidate solution has worse fitness than the previous then the candidate is rejected.

The overhead associated with the EHC is quite low on each iteration, requiring only a handful of elementary operations. This is worth consideration in order to reduce off-line training time.

3.4 Logarithmic Search

In order to present a deterministic approach to determining σ, we used a very simple logarithmic search algorithm (LOG). For a given search space (in this case, searching between a lower bound of 0 and and upper bound of 1.0), the logarithmic search works as shown in Algorithm 2:

Algorithm 2 Logarithmic Search

```
1:  procedure LOG
2:      range = upperbound − lowerbound
3:      while GRNN evaluations remain do
4:          step size = range / the number of steps
5:          for for k = 0 ... number of steps do
6:              σ = step size * k + lower bound
7:              determine fitness of σ
8:              store sigma of best fitness
9:              upper bound = best sigma + step size
10:             lower bound = best sigma - step size
11:             guarantee that new bounds are valid
12:             range = upper bound - lower bound
13:         end for
14:     end while
15: end procedure
```

The value of "steps" was chosen to be 200 in order to give sufficient granularity to find a (tiny) σ value. In plain English, the search divides a range into a number of discrete values and then tests all those discrete increments. Once the algorithm finds the best value in a range, it takes a smaller area around that value, subdivides it, and then performs the search again. This search continues until it either runs out of time or finds a performance within a given tolerance.

It is important to note that this search is perfectly adequate for a small search space, but it would not be scalable if the dimension of the search was increased to include more variables (which is part of the future goals of the system).

3.4.1 Steady-State Genetic Algorithm

In the initialization phase the SSGA [18] first generates a population of ten candidate solutions consisting of a binary listing of attributes (either "0" for not included in the classification or "1" for included in the classification) and a floating point value of sigma. Initial values of each candidate solution are determined by generating a uniformly distributed random value. In the second step these candidate solutions are each evaluated using the success rate of the GRNN on the validation set as fitness.

In the iteration phase the SSGA performs 1000 iterations of the next three steps. First, two binary tournament selections are used to select a first and second parent. These two parents are then used to create a child via uniform crossover. Second, there is a 10% probability of randomly flipping a single attribute bit in the child candidate solution in order to mutate the configuration. Finally, there is a random probability that a Uniform Mutation operator, is applied to the value of sigma, calculated as shown in Algorithm 3. The child will always replace the worst fit individual in the population. This process is repeated until the maximum number of allowed function evaluations has expired.

Algorithm 3 Mutate Sigma

```
1: procedure MUTATE SIGMA
2:     mutation_amount ← 0.125
3:     current_sigma ← current_sigma + rand(0, 1) ∗ mutation_amount ∗ current_sigma
4: end procedure
```

3.5 Statistical Algorithm

In order to be complete and we provide a statistical search algorithm to show that a statistical method is not an optimal solution. The input for the statistical search algorithm is a training set of files described as $F = \{F_0, F_1, \ldots, F_N\}$. Each file has a list of attributes to consider defined as $A = \{A_0, A_1, \ldots, A_M\}$. A_0 is a special attribute. It is the unique identifier describing the object being classified (e.g. compiler id) called the target_id. The statistical search algorithm works as follows:

First, determine the statistical information for the training set:

1. Calculate *sum* and *count* for each attribute for each target file. The *sum* is a map with the *key* equal to the target_id, *t*, and the *map value* equal to the list of attribute totals. The *count* is a map with the *key* equal to the target_id and the *map value* equal to the list of attribute count. The target_id is always the first attribute, a_0, associated with each file $f(a_0)$. This is shown in Algorithm 4.

Algorithm 4 Statistical Algorithm - Attribute Total and Count

```
1: procedure CALCULATE_COLUMN_TOTALS(F)
2:     for each file f ∈ F do
3:         t ← f(a_0)
4:         for a ← 1, n do                ▷ For each attribute a in the file minus target_id
5:             sum(target_id, a) ← sum(t, a) + f(a)
6:             count(target_id, a) ← count(target_id, a) + 1
7:         end for
8:     end for
9: end procedure
```

2. Calculate the average value, *avg*, for each attribute for each target. The *avg* is a map with the *key* equal to the target_id and the *map value* equal to the list of

the average attribute values. The first attribute, A_0, is skipped since this is the unique id for the type of object being classified. This is shown in Algorithm 5.

Algorithm 5 Statistical Algorithm - Attribute Average

```
1: procedure CALCULATE_COLUMN_AVERAGES(F)
2:     for t ← 0, size(sum) do                ▷ size(sum) gives the number of target ids
3:         for a ← 1, n do                    ▷ For each attribute minus the first
4:             avg(t, a) ← avg(t, a) + sum(t, a) / count(t, a)
5:         end for
6:     end for
7: end procedure
```

3. Calculate the variance value, *variance*, for each attribute for each target The *variance* is a map with the *key* equal to the target id, t, and the *map value* initially equal to the sum of the variance for each attribute values. Later the *map value* is equal to the variance for each attribute value for each target. This is shown in Algorithm 6.

Algorithm 6 Statistical Algorithm - Attribute Variance

```
1: procedure CALCULATE_COLUMN_VARIANCES(F)
2:     for each file f ∈ F do
3:         for each attribute a ∈ f do
4:             t ← first attribute in f
5:             variance(t, a) ← variance(t, a) + (f(t, a) − avg(t, a))^2
6:         end for
7:     end for
8: end procedure
```

4. Calculate the standard deviation for each attribute for each target. This is shown in Algorithm 7.

Algorithm 7 Statistical Algorithm - Attribute Standard Deviation

```
1: procedure CALCULATE_COLUMN_STANDARD_DEVIATIONS
2:     for t ← 0, size(sum) do                ▷ size(sum) gives the number of target ids
3:         for a ← 1, n do                    ▷ For each attribute minus the first
4:             stdev(t, a) += sqrt(variance(t, a))
5:         end for
6:     end for
7: end procedure
```

5. Calculate the lower bound and upper bound for each attribute for each target

3.5.1 Fitness Function

In order to evaluate the fitness of a candidate solution, we ran the GRNN with the indicated value of σ. Our first fitness criteria was the success rate (how often the GRNN correctly classified the file as self-modifying or not). In order to break ties in cases where the success rates for two candidate solutions were equal, the average distance from the desired output to the resultant output over all the test cases was used.

3.6 Summary

In this chapter, we described the machine learning technique, the inner workings of a General Regression Neural Network, and optimization techniques utilized in this research to optimize the parameters for the GRNN.

Chapter 4

Compressed Executables

In submission to the IEEE Transactions on Information Forensics and Security.

In order to effectively defend computers from malicious software, analysts need as much information as possible about files that will be executed on those machines. Particularly, analysts would like to have the capacity to make predictions about the danger posed by a file, but they would prefer to do so without actually executing the file (and therefore potentially doing harm to the system that the file is being run on). Thus, static file analysis is generally preferable from the perspective of the analyst. Our approach is to apply machine learning techniques to classify files with respect to their status of compression based on their Windows Portable Executable (Windows PE) header information. Specifically, we use General Regression Neural Networks upon a representative sample of training data to identify files with potentially self-modifying behavior. In the course of this work, we perform optimization for critical algorithm parameters. In order to accomplish these tasks, we make use of Evolutionary Hill-Climbing and a Logarithmic Search for parameter optimization. The system is tested on a previously unseen set of files to assess the ability of the system to generalize. Ultimately, this algorithm could be used to aid in anomaly detection mechanisms and forensic analysis by providing more information from the static analysis of executable files.

The remainder of the chapter is organized as follows: In Section 4.1, the specific experiments are described along with their results and discussion. Section 4.3 describes in detail the collection of data, the algorithms implemented for classification, and many of the important issues regarding these implementations. Section 4.4 addresses some general weaknesses of the approach. Section 15.2.7 describes continuing work with the

research, as well as future work for the Information Security community at large. Finally, a conclusion is given in Section 4.5.

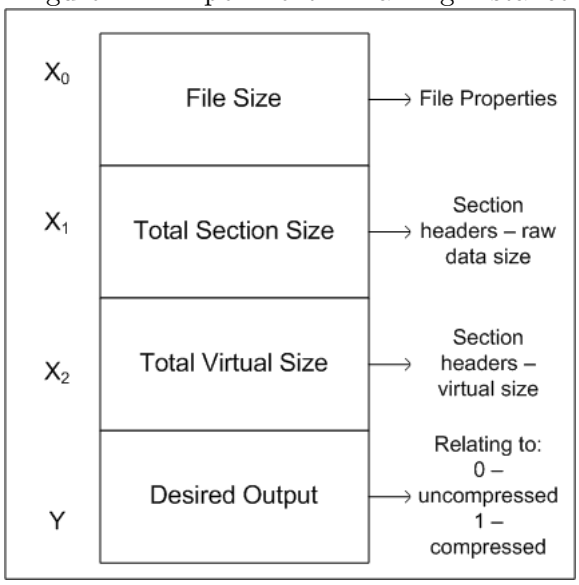
4.1 Experiments

In this section, the experiments are described, and the results are given, along with discussion of the implications of those results.

4.1.1 The Training, Validation, and Test Sets

A training instance includes three (3) fields of interest, (Total Size, Sum of Section Sizes, and Total Virtual Size) plus the desired classification value (see Figure 4.1). In Experiment I, the desired output was either a 0 (indicating uncompressed) or a 1 (indicating any type of compression).

Figure 4.1: Experiment I Training Instance



For Experiment II, the desired output varies from the set [0,1,2,3], where 0 indicates an uncompressed file, 1 represents a file produced by cexe, 2 represents a file compressed by PEtite, and 3 represents a file compressed by UPX (see Figure 4.2).

For Experiment III, the data was basically the same as Experiment II, but there are no uncompressed files (in other words, the only goal was to distinguish between the

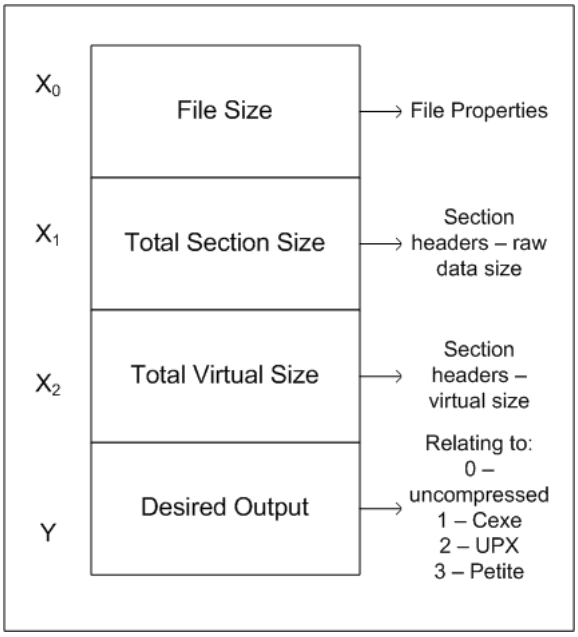


Figure 4.2: Experiment II Training Instance

type of compression used). In this case, the desired outputs vary from the set [0,1,2], where 0 indicates UPX, 1 represents Cexe, and 2 represents a PEtite file.

For all the above experiments, the training set consisted of roughly 600 file instances. The validation and test sets were collected with each containing approximately 70 random distinct unseen file instances. The only change in the sets between the first two experiments is the adjustment in the values for desired outputs (as described above). In the third experiment, however, the uncompressed files were removed from all sets, reducing their sizes by about 25%.

In each experiment, the training set was used as the training data for the GRNN. The optimization method was used on the validation set in order to optimize the GRNN. Then, the trained network was tested on the test set in order to provide our results. The rationale behind the three separate sets was to eliminate training bias resulting from the optimization of the GRNN.

Optimizer	Iterations to Best	Best Success Rate	Best σ	Mean Square Error
LOG	211	0.9143	4.14E-4	0.105
EHC	15.57	0.9143	4.14E-4	0.105

Table 4.1: Identifying Compression: GRNN Parameters and Results

4.1.2 Optimizer Stopping Conditions

The optimizers in all the experiments were given 1000 function evaluations (evaluations of the GRNN, which is the most expensive portion of the optimization, regardless of method).

4.1.3 Identifying Compression

The goal of this experiment was to classify an unseen file as either compressed or uncompressed (no attempt at this stage is made to determine what type of compression was used). First, we used the LOG optimization method (see Section 2) of determining the appropriate value of σ using the range [0,1.0].

The LOG approach (see Table 4.1) identifies a value of σ with a 0.9143 Success Rate (very high classification on unseen instances). This value of σ also yields a relatively low Average Distance of 0.105. While this approach is clearly deterministic, it carries a small penalty in terms of offline training time. The success of the search is bounded by the granularity of the step size (which may need to be quite small if the range of σ cannot be narrowed down somehow beforehand). In practice, as the number of training instances grows larger (which is desirable for greater ability to generalize over a wider variety of files), the cost to evaluate a GRNN will increase. In this case, evaluating the GRNN many times would be undesirable if training the network is done often in an actual system. To this end, we utilized the EHC to discover σ, as described in Section 3.3. It is worth noting that our goal is to produce a method for the quick, efficient, and effective discovery of σ, not merely for this specific problem, but rather for a general methodology that can be used for many file classification problems.

The EHC was run 30 times (for statistical significance) with 1000 maximum iterations on each run in order to discover a sigma value. The average solution (which always discovered the best known σ) is also provided in Table 4.1.

In Table 4.1, the value for "Iterations to Best" indicates the average number of iterations the EHC took to find its best value for σ (lower is better). The value "Best Success Rate" indicates the average percentage of correct identifications made on the testing set by the GRNN. The value "Best σ" indicates the mean of all the best-performing σ values. It should be noted that on every run, the EHC discovered the best value of σ. Finally, the "Mean Square Error" measure indicates the average distance between the desired output and the actual output given by the GRNN. These results show that the EHC, despite being a non-deterministic algorithm, still consistently identifies the best-performing values of σ. Further, it does so in a very small number of iterations, even fewer than the deterministic LOG optimizer.

The results for the first experiment were quite promising. Both the LOG and EHC methods of discovering σ ultimately resulted in very high Success Rates (with bests giving better than 90% classification). It should be noted, however, that the EHC method found the good σ values consuming less computational time. In general, the values of σ discovered were quite small, which seems to indicate that the training set is fairly representative of the classification space.

4.1.4 Classifying Type of Compression or Lack Thereof

Since Experiment I gave strong results the system was expanded to deal with the more difficult problem of identifying the compression utility used to compress a file (or to identify if the file was compressed at all). Again, first the LOG method was applied, as shown in Table 4.2. The value headings for all tables in this section have the same meaning as in Experiment I. The Best Success Rate discovered was 0.8857 (slightly lower than in the previous experiment). The σ values remain tiny, but the Best Average Distance roughly doubles from Experiment I. This is expected, however, given that there are a larger number of categories for classification.

Optimizer	Iterations to Best	Best Success Rate	Best σ	Mean Square Error
LOG	452	0.8857	2.42E-5	0.214
EHC	58.8	0.8857	2.5E-5	0.214

Table 4.2: Classifying Type of Compression or Lack Thereof: GRNN Parameters and Results

Optimizer	Iterations to Best	Best Success Rate	Best σ	Mean Square Error
LOG	452	0.9149	2.42E-5	0.107
EHC	36.57	0.9149	2.42E-5	0.107

Table 4.3: Classifying Type of Compression: GRNN Parameters and Results

Again, the EHC was also applied to σ search problem. The results in Table 4.2 show the average performance on the search problem. The Best σ is discovered in an average of 58.8 EHC iterations and Average Success Rate is the same as the results discovered through the LOG optimization approach.

In general, the results from Experiment II are quite promising. Even when dealing with a more complex classification problem, the GRNN *with the same training data* still managed to effectively classify the type of compression with success rates of nearly 0.89.

4.1.5 Classifying Type of Compression

In the third set of experiments, the system was tasked with determining type of compression in the absence of uncompressed files to see if having *a priori* information regarding the files could improve accuracy. It is conceivable that files to be tested are known to be compressed already (eliminating the possibility of the network confusing files for uncompressed), which could increase the accuracy of the classifications. Essentially, this experiment deals with detecting the differences between the properties in files known to be compressed with differing algorithms.

Again, both the LOG and EHC optimization methods were applied, as shown in Table 4.3. The Best Success Rate discovered was 0.9149 (better than in the previous experiment) for both optimizers. In fact, they perform roughly as well as the classifiers in Experiment I. In general, both algorithms discover the best σ, but the EHC does so faster.

4.1.6 Impact of Compression

While certain attributes, such as the capacity for self-modification, compression, and encryption may make an executable suspicious, all of these capabilities have legitimate purposes as well [69]. For example, completely reasonable uses of compression include:

- Software on embedded systems
- NASA Mars Exploration Rovers Spirit and Opportunity used on-board loss-less data compressors
- to deter reverse engineering
- to obfuscate contents of executables
- programs designed for portability on resource-poor media
- games with large quantities of data
- large applications available for download
- backups are sometimes saved as compressed executables to save disk space

The lesson taken from the various uses of compression demonstrates that identifying a file as compressed is only one part of an analysis. However, knowing the compression type of a file makes the analysis of said file considerably easier (or sometimes makes an infeasible problem feasible). It should be noted that a file should typically only be flagged as "risky" when it exhibits multiple symptoms.

Noting the above, file compression and encryption have a negative impact on the capability of many schemes to analyze executable files. Many methods for detecting malicious programs, especially for virus and malware, depend on specific "signatures" composed of a specific group of bytes to match files. Such signatures are very specific in nature (to avoid false positives), but this makes them highly ineffective against even small alterations in the executable. Signature-based schemes have reduced capacity to

identify a file if that file's corresponding signature is compressed or encrypted. Worse still, in order to address such a problem, the signature-detecting scheme would need a corresponding signature for *every* possible encryption and compression methodology that might produce different output! Clearly, given the vast number of encryption and compression algorithms, this is simply not feasible (the signature database would swell to epic proportions).

4.2 Data source and compressors

To get a representative sampling of file types and sizes we used a large number of files, including wide variety of executables, from self-written executables to common applications. We then used three different compression utilities, Cexe, Ultimate Packer for eXecutables (UPX) [46], and PEtite [19] to compress these files. There were a few instances where one of the compression utilities failed to compress the original executable; however, the information from the remaining utilities' compression of the same file was still included.

4.2.1 Cexe

Cexe is a Win32 compressor that tries multiple compressors and chooses the one that gives the smallest size. The version that we used contained compressors for Lempel-Ziv-Welch (lzw), published in 1984 [67], and zlib, which was written by Jean-loup Gailly. Often zlib was used; however, there were some of the smaller files that Cexe used lzw to compress.

4.2.2 Ultimate Packer for eXecutables

UPX, written by Markus F.X.J. Oberhumer, Làszló Molnàr and John F. Reiser, is a packer for executables of several different formats. It uses the Not Really Vanished (NRV) compression library, a generic data compression library written by oberhumer.com, a small company in Linz, Austria.

4.2.3 PEtite

PEtite is another executable packer which compresses an executable while still allowing the executable to run exactly as the original uncompressed version after loading. While we could not get information on exactly which compression is used by PEtite, after viewing the compression ratios, the method of the compression that is utilized is different from both Cexe and UPX.

4.3 Data Extraction

We created a TrainerDump program which operates on a similar basis as the popular Pedump utility to extract PE file information. It uses the Libreverse library for opening the Windows PE32 executables in memory and then parsing its headers. The MS-DOS header is read to determine if the program under observation is in fact a windows executable. The file extension does not guarantee any particular property about an executable file. The MS-DOS header also provides the location of the Windows PE header which tells us where to look for the section headers. In the Windows PE section header, we can find two of the properties that are required for the trainer: virtual memory size of the section when loaded and its raw size in the binary file on disk. For each section header found on the disk, a total is produced for both along with the actual size of the file.

The actual size of the file is an obvious fact about a binary program. Typically, the binary program takes as much space as required. There is little reason to pad the file so the size is larger on the system. This is contradictory to what a malicious program designer wants to achieve. They often want their application to be as small as possible to avoid suspicion. The data that can potentially be altered are the virtual memory size and the raw data size. The virtual memory size is what is required to store the data or instructions in memory. If the value is smaller than the size of raw data on disk, then the loader will have a memory error when it attempts to put data past the end of the space. If the memory request is equal to or larger than the size of raw data, the loader

can perform the loading without errors. The virtual memory size must be at least as large as the raw data size. Most virus writers can alter this number. It is most often altered to ensure that the sufficient space is available for decompressing/decrypting a real binary program. Finally, the raw data size cannot be less than the entire program. If a malicious program has a stated raw data size, it is safe to conclude that the size is at least equal to or greater than the actual instructions. Just because the bytes in a section are marked as executable, does not mean that they will actually be used. So the raw data size can be larger than is required, but this is a negative feature if we are trying to minimize our file size.

TrainerDump will be reading the target file so the access time of the target file will change. It is important to also note that none of the bytes of the malicious executable are altered or executed. This is necessary when considering that forensic tools should not modify a suspect file during analysis. Therefore, the PE files should have read-only set on them just as a safety back up. TrainerDump program extracts the necessary facts for a large number of files into a space delimited file for use by the GRNN.

4.4 Weaknesses

Our approach carries some practical weaknesses that should be noted in the name of good scholarship. First, while our GRNN proved to be an effective classifier, it is dependent on the quality of the training set. If such a training set could not be engineered (perhaps the information used for the training instances is not available for some type of executables), then this strategy would not be useful. Second, we rely on the information provided in the PE Header to perform the classification. If this header could be distorted or misrepresented by a malicious entity, then it could easily fool our system by presenting the file's pre-compressed header information. Third, training instances must be gathered for every file type that may be subject to analysis. This returns to a fundamental problem regarding static analysis of executables in that they

tend to be system specific. This may not be a significant problem, since most reverse-engineering efforts are already very system-specific (meaning that this restriction is no greater than those that already exist).

4.5 Summary

In this chapter, we motivated the need for effective malware analysis upon executable files when neither the source code is readily available nor is there any desire to execute the file to examine its behavior. Further, we implemented logarithmic search and evolutionary search tactics in order to optimize the system parameter (σ) of the GRNN. The fairly promising results of the system were discussed and, finally, we provided a number of opportunities for the expansion of this work.

Chapter 5

A Compiler Classification Framework for Use in Reverse Engineering

5.1 Introduction

In submission to IEEE Symposium Series on Computational Intelligence in Cyber Security (CICS 2009)

A software framework is presented in this chapter for extracting useful information from Java Class, Windows PE and Linux ELF files and analyzing that information to classify future files. A General Regression Neural Network is implemented and optimized using both deterministic and stochastic search algorithms. In experimental results, the system can classify compiler type on an unseen file with a more than a 98% degree of accuracy.

5.1.1 How can machine learning be used for classifying compiler type?

The problem of determining if a binary computer application was compressed or not is quite similar to the problem of classifying the compiler used to create a binary computer application. Both require certain attributes from the header of the binary computer application in order to make a determination. In order to perform the analysis, first information is extracted from the files. Instead of selecting attributes from the files to determine compression, new attributes would have to be selected. Once the training data is collected, a General Regression Neural Network (GRNN) [53] learns from the training data and attempts to estimate the type of compiler used to create a file previous unseen by the GRNN.

Once a training set consisting of the file summaries is constructed along with the GRNN, two different optimization methods are used to optimize the neural network parameters. This is similar to the ideas suggested in [22] for the optimization of weights

in a Feed-Forward Neural Network using a Particle Swarm Optimizer. Experiments were developed to test the effectiveness of the system on classifying unseen file instances and recording metrics of success rate and average distance to desired output. The structure of the training set and the network parameters were adjusted for the problem of classifying which of three families of popular modern Java Compilers, three families of popular modern C++ compilers for Linux or three families of popular modern C++ compilers for Windows were used to compile a given set of test files.

Java	JDK v5.0 JDK v6.0 IBM Jikes ECJ
Windows	Cygwin GCC 4.1 Visual Studio 2003 Visual Studio 2005
Linux	GCC 4.1 Portland Group C++ 7.1 Intel C++ 10.1

Table 5.1: List of compilers

5.2 Software Architecture

In this section, we describe the software architecture of the compiler classifier and the XML format of the input.

5.2.1 Java Input Data

The input information used by the compiler classifier was obtained from a simple program which utilizes the Libreverse library to read the input Java Class files. These input training sets of Java Class files were produced by compiling a set of test Java files with the Eclipse, Jikes, Sun Microsystem 1.5 and 1.6 Java compilers. The Libreverse library is a reverse-engineering architecture developed to automatically produce the desired high-level output from a given input based on predefined configurations of analysis components.

The input data was formatted in XML using the schema shown in Listing 5.1 in order to produce a easy parse input format to the classifier.

Listing 5.1: XML schema for input Java data

```
<?xml version=''1.0''?>
<xs:schema>
  <xs:element name=''data''>
    <xs:complexType>
      <xs:sequence>
        <xs:element name=''file'' type=''file_type''/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name=''file_type''>
    <xs:sequence>
      <xs:element name=''target_id'' type=''xs:decimal''/>
      <xs:element name=''size'' type=''xs:decimal''/>
      <xs:element name=''version'' type=''xs:decimal''/>
      <xs:element name=''this_class_index'' type=''xs:decimal''/>
      <xs:element name=''super_class_index'' type=''xs:decimal''/>
      <xs:element name=''constant_pool_count'' type=''xs:decimal''/>
      <xs:element name=''constant_pool_info''
                  type=''constant_pool_type''
                  minOccurs=''12'' maxOccurs=''12''/>
      <xs:element name=''field_count'' type=''xs:decimal''/>
      <xs:element name=''method_count'' type=''xs:decimal''/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name=''constant_pool_type''>
    <xs:sequence>
      <xs:element name=''tag'' type=''xs:decimal''/>
      <xs:element name=''count'' type=''xs:decimal''/>
      <xs:element name=''ratio'' type=''xs:decimal''/>
    </xs:sequence>
  </xs:complexType>
<xs:schema>
```

The description of each of the elements are:

- **compiler_id**: Unique id associated with the compiler that produced the file.
- **size**: File size
- **version**: This value is used to indicate what release level of Java this binary requires the JVM to support.

- **this_class_index**: The value of the this_class item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a Constant Class Info structure representing the class or interface defined by this class file [8].
- **super_class_index**: This is similar to the this_class defined above but the only difference is that this index refers to a parent class or the default Object class.
- **constant_pool_count**: Number of constant pool entries.
- **constant_pool_info (tag)**: Unique id for the constant pool entry.
- **constant_pool_info (count)**: Number of times the unique id was seen.
- **constant_pool_info (ratio)**: Ratio of the number times to the number of constant pool entries.
- **field_count**: Number of field_info structures in the class file.
- **method_count**: Number of method_info structures in the class file.

5.2.2 ELF Input Data

The input information used by the compiler classifier was obtained from a simple program which utilizes the Libreverse library to read the input ELF files. These input training sets of ELF files were produced by compiling a set of test C++ files with the GNU GCC 4.1, Portland Group C++ 7.1, and Intel C++ 10.1 compilers. The Libreverse library was used to automatically produce the desired high-level output from a given input based on predefined configurations of analysis components.

The input data was formatted in XML using the schema shown in Listing 5.2 in order to produce a easy parse input format to the classifier.

The description of each of the elements are:

- **compiler_id**: Unique id associated with the compiler that produced the file.
- **size**: File size

Listing 5.2: XML schema for input ELF data

```
<?xml version=''1.0''?>
<xs:schema>
  <xs:element name=''data''>
    <xs:complexType>
      <xs:sequence>
        <xs:element name=''file'' type=''file_type''/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name='file_type'>
    <xs:sequence>
      <xs:element name=''target_id'' type=''xs:decimal''/>
      <xs:element name=''size'' type=''xs:decimal''/>
      <xs:element name=''entry_point_address'' type=''xs:decimal''/>
      <xs:element name=''section_headers_start'' type=''xs:decimal''/>
      <xs:element name=''program_header_count'' type=''xs:decimal''/>
      <xs:element name=''section_header_count'' type=''xs:decimal''/>
      <xs:element name=''section_header_string_table_index''
                  type=''xs:decimal''/>
      <xs:element name=''text_section_size'' type=''xs:decimal''/>
    </xs:sequence>
  </xs:complexType>
<xs:schema>
```

- **entry_point_address**: Virtual address where the system first transfer control.
- **section_header_start**: Where the Section header table is located in the file.
- **program_header_count**: Number of entries in the program header table.
- **section_header_count**: Number of entries in the section header table.
- **section_header_string_table_index**: Section header table index associated with the string table.
- **text_section_size**: Size in bytes of the executable code section in memory.

5.2.3 Windows PE Input Data

The input information used by the compiler classifier was obtained from a simple program which utilizes the Libreverse library to read the input Windows PE files. These input training sets of Windows PE files were produced by compiling a set of test C++

files with the GNU GCC 4.1 (Cygwin), Visual Studio 2003 and 2005 C++ compilers. The Libreverse library was used to automatically produce the desired high-level output from a given input based on predefined configurations of analysis components.

The input data was formatted in XML using the schema shown in Listing 5.3 in order to produce a easy parse input format to the classifier.

Listing 5.3: XML schema for input PE data

```
<?xml version=''1.0''?>
<xs:schema>
  <xs:element name=''data''>
    <xs:complexType>
      <xs:sequence>
        <xs:element name=''file'' type=''file_type''/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name='file_type'>
    <xs:sequence>
      <xs:element name=''target_id'' type=''xs:decimal''/>
      <xs:element name=''size'' type=''xs:decimal''/>
      <xs:element name=''exe_header_address'' type=''xs:decimal''/>
      <xs:element name=''coff_section_header_count''
                  type=''xs:decimal''/>
      <xs:element name=''pe_opt_code_size'' type=''xs:decimal''/>
      <xs:element name=''pe_opt_base_of_data'' type=''xs:decimal''/>
      <xs:element name=''pe_opt_entry_point'' type=''xs:decimal''/>
      <xs:element name=''pe_opt_image_size'' type=''xs:decimal''/>
    </xs:sequence>
  </xs:complexType>
<xs:schema>
```

The description of each of the elements are:

- **target_id**: Unique id associated with the compiler that produced the file.
- **size**: File size
- **exe_header_address**: File offset where the Windows PE header starts in the file header.
- **coff_section_header_count**: Number of section headers in the file header.
- **pe_opt_code_size**: Size of the code section.

- **pe_opt_base_of_data**: Base address of the data section.
- **pe_opt_entry_point**: Address of the first instruction to be executed.
- **pe_opt_image_size**: Size of the loaded image in memory.

5.2.4 Structure

There are two basic sections to the compiler classifier: Input and Analysis. The Input section is responsible for parsing the XML files, preparing the data structures, separating data into training, test and verification sets.

The input data is retrieved from the input XML files by the IO class. It is responsible for calling the Training_Data_Parser for each input file and then processing it with the Selection_Policy to get the final Training_Set. A Training_Set is used to contain the three types of data (training, test and verification).

The main function of the classifier optimizer takes the data retrieved from the IO class and passes it to the Optimizer. The Optimizer is the part that coordinates the processing of the input data. Processing continues until the number of iterations is achieved or time runs out. At that point the success value, final sigma value, and attributes used are output from the program.

5.3 Experiments

In this section, the three major experiments are described and the results of said experiments are given, along with discussion.

5.3.1 The Training, Validation, and Test Sets

A training or test or validation instance potentially includes many fields of interest.

In Experiment I (Java Compiler Classification with the GRNN Optimized with an Evolutionary Hill-Climber Algorithm), a simple optimization was utilized to try to determine appropriate attributes and system parameters for the GRNN. There were a total of 29 Java attributes available plus the continuous variable σ to be optimized. The

Compiler	ID
ECJ	0
Jikes	1
JDK v5.0	2
JDK v6.0	3

Table 5.2: Java classification values

Compiler	ID
GCC 4.1	0
Portland Group C++ 7.1	1
Intel C++ 10.1	2

Table 5.3: ELF classification values

goal was to create a GRNN that would be most accurate at classifying the compiler type to a value in Table 5.2 for unseen Java file instances. Additionally, potentially reducing the number of attributes used by the GRNN was motivated by:

- Given that not all of the attributes are strongly correlated with compiler type, some of this information may actually detract from the GRNN's ability to classify correctly.
- Reducing the size of the instances greatly improves the speed of the classifier at run time
- A consequence of reducing run time speed of the classifier is that the time required to perform optimization is dramatically reduced.

For Experiment II (Java Compiler Classification with the GRNN Optimized with a Steady-State Genetic Algorithm), the experimental goals were unchanged, but a more sophisticated optimization routine (a genetic algorithm) was used to optimize the attributes and system parameters.

For Experiment III (ELF File Classification with the GRNN Optimized with a Steady-State Genetic Algorithm), the experimental setup was essentially the same as in Experiment II, with the exception that ELF file data was being used to classify an unseen file to a value in Table 5.3.

Compiler	ID
Cygwin GCC 4.1	0
Visual Studio 2003	1
Visual Studio 2005	2

Table 5.4: PE classification values

For Experiment IV (Windows PE File Classification with the GRNN Optimized with a Steady-State Genetic Algorithm), the experimental setup was essentially the same as in Experiment II and III, with the exception that Windows PE file data was being used to classify an unseen file to a value in Table 5.4.

It should be noted that using a deterministic optimization approach in order to determine the attributes in this case would be more time-consuming since the search space grows by a factor of 2^n, where n is the number of attributes (in this case 29 for Java and 10 for the ELF files).

For all of the Java experiments, the training set consisted of roughly 334 class file instances, collected as mentioned in Section 5.2.1. The validation and test sets were collected in the same manner each containing approximately 50 - 70 random distinct unseen class file instances.

For all of the Linux ELF Experiments, the training set consisted of 567 file instances with validation and test sets containing about 114 random distinct unseen file instances.

For all of the Windows PE Experiments, the training set consisted of 563 file instances with validation and test sets containing about 112 random distinct unseen file instances.

In each experiment, the training set was used as the training data for the GRNN. The optimization method was used on the validation set in order to optimize the GRNN. Then, the trained network was tested on the unseen test set in order to provide our final results. The rationale behind the three separate sets was to eliminate training bias resulting from the optimization of the GRNN. In other words, the optimization process does not see the test set instances at any time. Only after the completion of training is the test set used.

	σ	Success Rate	Mean Square Error
min.	0.00181	54.5%	0.050
max.	0.00242	95.1%	0.964
stdev.	9.41E-05	9.9%	0.199
avg.	0.00200	78.3%	0.285

Table 5.5: Experiment I: Identifying Java Compiler Type with GRNN optimized by the EHC. Shows the results of optimizing a GRNN using an Evolutionary Hill-Climber Algorithm. The EHC was allowed a maximum of 10000 iterations, had a 10% chance of modifying σ each iteration, and always performed a single bit-flip mutation on a random attribute.

5.3.2 Optimizer Stopping Conditions

All the optimizers in all the experiments were given a fixed number of function evaluations (evaluations of the GRNN, which is the most expensive portion of the optimization, regardless of method). This cap on optimization time is described in the experiment.

5.3.3 Experiment I: Java Compiler Classification with a GRNN Optimized with an Evolutionary Hill-Climber Algorithm

For this experiment set, the Evolutionary Hill-Climber was used as the optimization method (with settings as described in Section 3.3) to distinguish between the four Java compiler types.

5.3.3.1 Distinguishing Four Compiler Types

The goal of this experiment was to classify the unseen class files (the test set) into the appropriate compiler type, using all four possible types.

It is worth noting that our goal is to produce a method for the quick, efficient, and effective discovery of σ not merely for this specific problem compiler classification problem, but for a general methodology that can be used for many file classification problems.

The EHC was run 30 times (for statistical significance) maximum of 10000 iterations on each run in order to discover a sigma value and a good combination of attributes.

The experimental results are provided in Table 5.5. In the table, the value indicated by "σ" represents best "σ" value discovered by the EHC. The "Succeses Rate" indicates the number of correct classifications by the configuration on unseen test data. A higher success rate is better. Finally, the "Mean Square Error" as defined as average squared difference between the predicted value and actual value. In other words, we favor classifiers that not only correctly identifier the compiler type, but also that give a continuous value that is as close as possible to the actual value. A lower Mean Square Error is better.

In general, it should be noted that the performance on this problem was weaker than could be used in practice. While the EHC finds a solution quickly (on average, in only a few iterations), this is likely due to premature convergence to a local minima. The average values of σ are relatively small (hovering around 2E-03) with only a small degree of variability. The average success rate discovered is about 78%, with around 10% standard deviation. This much error and variability indicates that this particular strategy would be undesirable for everyday application. Not surprisingly, the Mean Square Error is also fairly high, with an average of 0.285.

While these results are somewhat promising in that the optimized classifier clearly does much better than random guessing (which would be expected to perform at about 25% accuracy), it still does not achieve a low enough error rate to be used in a practical setting. The weakness in optimization seems to be the inability of the EHC to break out of local minima, which means that it never finds the best configuration of attributes. In order to fix this problem, a more powerful search mechanism, the SSGA, was used.

5.3.4 Experiment II: Java Compiler Classification with a GRNN Optimized with a Steady-State Genetic Algorithm

Since Experiment I gave weaker than desired results, a more powerful optimization algorithm was used. The primary hope was that this optimization would be better able to remove data that had a weak correlation to the compiler type and thus improve the relatively weak success rates obtained so far. This seems reasonable because the

	σ	Success Rate	Mean Square Error
max.	0.00233	100.00%	0.159
min.	0.00177	90.24%	0.00190
avg.	0.00201	96.04%	0.0599
stdev.	0.00011	2.85%	0.0457

Table 5.6: Experiment II: Identifying Java Compiler Type with GRNN optimized by the SSGA. Shows the results of optimizing a GRNN using a Steady-State Genetic Algorithm. The SSGA was allowed a maximum of 100 iterations, the population size was set to be 10, the mutation amount was set to be 0.125, and the mutation rate was set to be 0.10.

	σ	Success Rate	Mean Square Error
max.	0.00246	100.00%	0.152
min.	0.00172	94.51%	4.64E-12
avg.	0.00199	98.53%	0.0323
stdev.	0.00017	1.48%	0.0338

Table 5.7: Experiment II: Identifying Java Compiler Type with GRNN optimized by the SSGA. Shows the results of optimizing a GRNN using a Steady-State Genetic Algorithm. The SSGA was allowed a maximum of 1000 iterations, the population size was set to be 10, the mutation amount was set to be 0.125, and the mutation rate was set to be 0.10.

crossover operation (as described in Section 3.4.1) has the potential to change many attributes in the configuration compared to the one attribute that could be changed by random bit-flip mutation in the EHC. The same training, validation, and test data from the Java files used in Experiment I was used in Experiment II. The settings for the SSGA itself are as described in Section 3.4.1.

The results from this experiment allowing only 100 function evaluations (instead of the 10000 which were allowed for the EHC) are shown in Table 5.6. The results for the same experiment allowing 1000 function evaluations are shown in Table 5.7. The meanings of the column headings is unchanged from Experiment I.

While the number of required iterations needed to find the best solution increases in both cases, this seems to be attributed to the fact that those iterations are consistently finding better configurations as opposed to getting stuck in local minima. Even when the optimizer is limited to 100 iterations, the average success rate is still notably better at 96.04%, albeit with a higher standard deviation of 2.85%. However, in the 1000 iterations case, the success rate achieved rises to 98.53% with a much lower standard

deviation of 1.48%. This indicates a level of accuracy useful for practical application. In both the 100 iteration and the 1000 iteration case, the average Mean Square Error drops, garnering 0.0457 and 0.0338, respectively. The average discovered value of σ remains around 2.01E-03, which seems to indicate that this is a good value of σ for this problem.

It should be noted that the dramatically improved accuracy makes this approach plausible for practical application. Although not visible from the above charts, the vast majority of the classification error stems from the GRNN's inability to distinguish well between the two versions of the JDK. Again, this is due to the fact that the attributes chosen change very little between these two versions.

5.3.5 Experiment III: ELF File Compiler Classification with a GRNN Optimized with a Steady-State Genetic Algorithm

Given the promising results of the approach in Experiment II, a similar strategy was pursued for the Linux ELF files. Different attributes were available for selection (as described in Section 5.2.2), but the same SSGA and GRNN combination was used. Since the EHC results were not very promising in Experiment I, only the SSGA was used for Experiment III. Again, the optimizer was first allowed 100, then 1000 iterations in order to discover an optimal configuration. The results from this experiment are reported in Table 5.8 and Table 5.9. A run of the optimizer, described in Section 6.3.3, can have a iteration value of 0. This is due to the fact that the best solution found will have already existed before any children were considered.

The results for the ELF files were quite good, showing again results of practical value. Even when given only 100 iterations, a configuration with 99.51% success rate is discovered. Further, not much additional value is gained from further training (only a small improvement in terms of Mean Square Error in the 1000 iteration case). Even in the worst case run, a configuration with 97.21% success rate is discovered. In the best case, the success rate is 99.51%.

	σ	Success Rate	Mean Square Error
max.	0.00224	100.00%	0.0561
min.	0.00176	97.21%	5.53E-011
avg.	0.00199	99.51%	0.0128
stdev.	0.0001	0.69%	0.0179

Table 5.8: Experiment III: Identifying C++ Compiler Type for Linux ELF with GRNN optimized by the SSGA. Shows the results of optimizing a GRNN using a Steady-State Genetic Algorithm. The SSGA was allowed a maximum of 100 iterations, the population size was set to be 10, the mutation amount was set to be 0.125, and the mutation rate was set to be 0.10.

	σ	Success Rate	Mean Square Error
max.	0.00222	100.0%	0.04
min.	0.00179	98.60%	0.00
avg.	0.00198	99.88%	0.00191
stdev.	7.84E-05	0.32%	0.00740

Table 5.9: Experiment III: Identifying C++ Compiler Type for Linux ELF with GRNN optimized by the SSGA. Shows the results of optimizing a GRNN using a Steady-State Genetic Algorithm. The SSGA was allowed a maximum of 1000 iterations, the population size was set to be 10, the mutation amount was set to be 0.125, and the mutation rate was set to be 0.10.

Ultimately, this experiment shows that the approach has promise in terms of being able to produce good results for a wide variety of file types on differing platforms.

5.3.6 Experiment IV: Windows PE File Compiler Classification with a GRNN Optimized with a Steady-State Genetic Algorithm

Given the strong results of the approach in Experiment II and III, a similar strategy was pursued for the Windows PE files. Different attributes were available for selection (as described in Section 5.2.3), but the same SSGA and GRNN combination was used. Since the EHC results were not very promising in Experiment I, only the SSGA was used for Experiment IV. Again, the optimizer was first allowed 100, then 1000 iterations in order to discover an optimal configuration. The results from this experiment are reported in Table 5.10 and Table 5.11.

	σ	Success Rate	Mean Square Error
max.	0.002	100.00%	0.0356
min.	0.00192	99.11%	0.00
avg.	0.002	99.77%	0.00820
stdev.	1.45E-05	0.37%	0.0145

Table 5.10: Experiment III: Identifying C++ Compiler Type for Windows PE with GRNN optimized by the SSGA. Shows the results of optimizing a GRNN using a Steady-State Genetic Algorithm. The SSGA was allowed a maximum of 100 iterations, the population size was set to be 10, the mutation amount was set to be 0.125, and the mutation rate was set to be 0.10.

	σ	Success Rate	Mean Square Error
max.	0.00222	100.0%	0.0400
min.	0.00179	98.60%	0.00
avg.	0.00198	99.88%	0.00191
stdev.	7.84E-05	0.32%	0.00740

Table 5.11: Experiment III: Identifying C++ Compiler Type for Windows PE with GRNN optimized by the SSGA. Shows the results of optimizing a GRNN using a Steady-State Genetic Algorithm. The SSGA was allowed a maximum of 1000 iterations, the population size was set to be 10, the mutation amount was set to be 0.125, and the mutation rate was set to be 0.10.

5.4 Weaknesses

With respect to the methodology, it is important to realize that machine learning is dependent on the quality of the training data. If the training data is unrelated to the compiler type or if there is insufficient training data available, then the machine learner will perform poorly (regardless of optimization). It should also be noted that the offline training becomes time-consuming when the appropriate attributes to use as input are unknown (which is the expected situation). Further, optimization will have to be repeated for every new type of classification problem (although this process can be largely automated as has been demonstrated above).

Currently, the training data is extracted from the files, but, if certain fields could somehow be altered significantly, then the classifier would have diminished value. This might prove to be a difficult task, but, if a file was being designed to thwart reverse engineering efforts, then it is not outside of the realm of possibility.

5.5 Summary

In this work, we provide a general motivation for reverse engineering of files, as well as the value added by identifying attributes about those files from the Java Class, Linux ELF and Windows PE files themselves. In doing so, we motivate the need for the ability to identify the compiler used to produce a Java Class, Linux ELF or Windows PE file. Moreover, this identification should take place with only knowledge obtained from the target file itself. We present a software system capable of extracting information from Java Class, Linux ELF and Windows PE files and analyzing those files appropriately and accurately. This analysis makes use of a General Regression Neural Network that is optimized using a pair of optimization techniques. Experiments were performed on a large test set of Java Class, Linux ELF and Windows PE files to validate the approach and classification error was shown to be less than 2%. Ultimately, the contribution of this work is to contribute to reverse engineering by providing a means to identify the type of compiler used to compile a Java Class, Linux ELF and Windows PE file using only information obtained from the target file itself.

Chapter 6

Classifier Architecture

The compiler classifier is a three stage process. The first stage, offline file processing, is responsible for gathering the desired file properties from the input executable produced by a compiler for a certain file type (e.g. Linux ELF). These values are stored into a XML file associated with the compiler for use by the optimizer. The second stage, offline optimization, is responsible for taking the input XML files for a set of compilers and processing them through a GRNN. The output of the second stage is a sigma value, a filtered XML file containing the actual file properties used and a list of attributes used for classifying a executable file for that file type (e.g. Linux ELF). The final stage, on-line compiler classification, is responsible for classifying the compiler used to create a unknown file by processing the saved XML data and the unknown file properties through a GRNN. The on-line processing will save the detected compiler id as Meta information for use by other child Components.

6.1 Offline File Processing

The most time intensive part of classification is the optimization phase. It is during this part of the classification that the actual values used for classifying new files are determined. The offline file processing is started first by collecting the values using the compiler dump program, described in Section 6.1.1 and then Section 6.3.3.

6.1.1 Compiler Dump Program

The compiler dump program was created to collect the attributes that are believed by an human expert to be necessary to classify the compiler. It is important to understand that the actual attributes necessary to classify the compiler will not be known

until after the optimizer program is executed on the collected data. Each of the attributes collected are believed to be either difficult to falsify or counterproductive if they are altered. Attributes that are "informative" in nature, for example the date the target program was created, can either be ignored or eliminated without affecting the program's execution. The compiler dump program requires four pieces of information from the user via command line flags:

- **compiler name**: String name to be associated with the compiler that produced the set of test programs.
- **compiler id**: Unique integer identifier to be associated with the compiler that produced the set of test programs.
- **source type**: String name describing the type of test programs (e.g. Java class).
- **directory**: Location where the test programs can be found.

The compiler dump program uses these command line flags to locate the appropriate algorithm to use, collect the test programs from the given directory, process each executable program, and store the collected information in a XML file. Section 6.1.1.1 will describe each of the kind of algorithms available. Section 6.2 describes the XML schema associated with the type of test programs. Figure 6.1 shows the structure of the compiler dump program.

6.1.1.1 Compiler Dump Algorithm

Each algorithm that is used to gather suspect file attributes must follow the interface laid out in the Compiler Dump Algorithm class. The use of an abstract base class for the Compiler Dump Algorithm was necessary since it is important to hide the kind of algorithm being used from the trainer program. It is unnecessary for the compiler dump program to know which algorithm it is interacting with when processing a set of files. This property allows the design to support a wide variety of file types and allows scalability to future design changes.

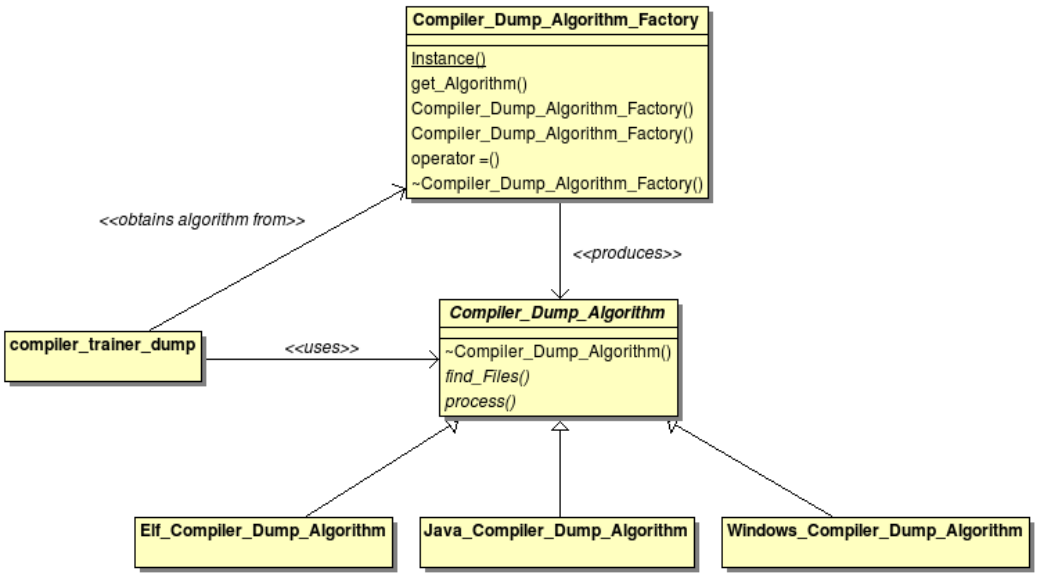


Figure 6.1: Compiler Dump Algorithm class diagram

The correct Compiler Dump Algorithm is found through the Compiler Dump Algorithm Factory's *get_Algorithm* function. The Compiler Dump Program calls this function, obtains the correct algorithm, and then calls the algorithm's functions to perform its work.

6.1.1.1.1 ELF Compiler Dump Algorithm The ELF Compiler Dump Algorithm, as shown in Figure 6.2, is used when processing a Linux ELF executable. Since the advent of 64-bit computing the ELF format was extended to support 64-bit executables. Therefore the ELF Compiler Dump Algorithm must first determine which type of file is being read, load the file via the correct file reader, and then read the necessary attributes. This is shown in Figure 6.3.

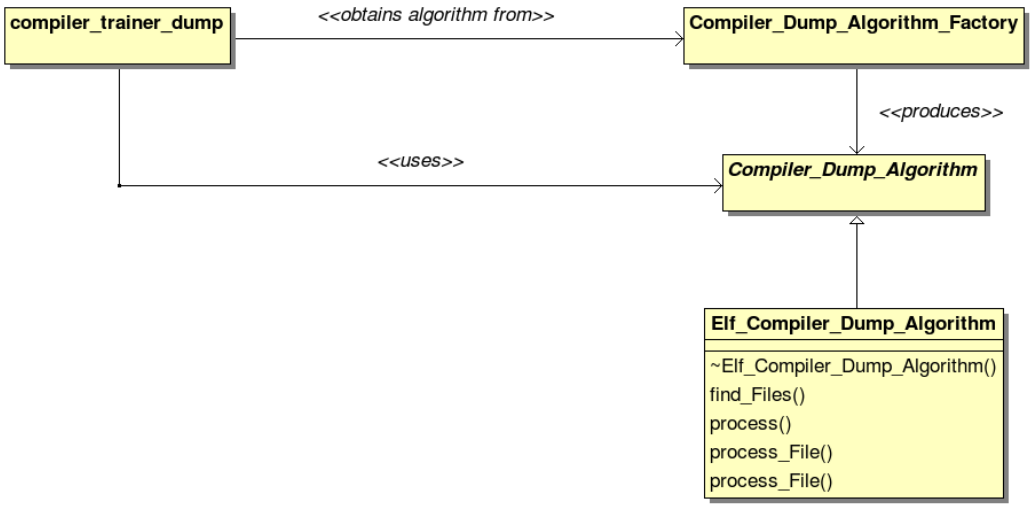


Figure 6.2: ELF Compiler Dump Algorithm class diagram

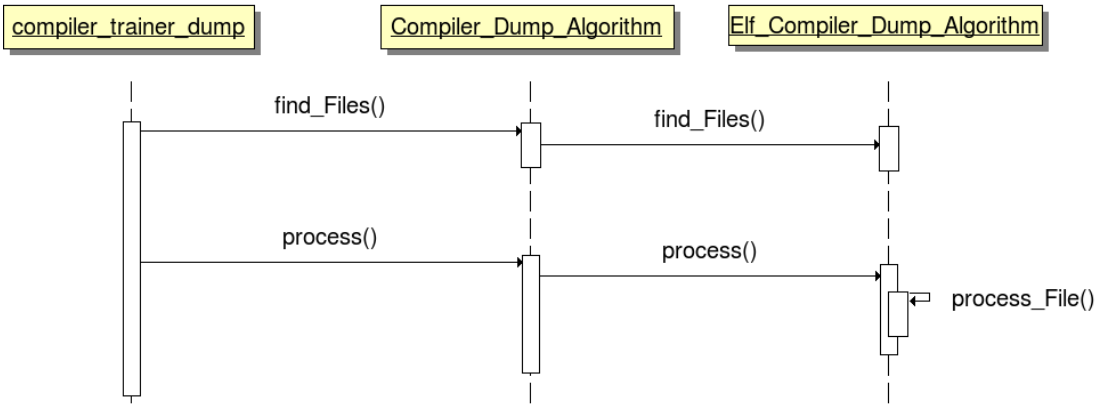


Figure 6.3: ELF Compiler Dump Algorithm sequence diagram for reading a 32-bit Linux ELF file

6.1.1.1.2 Windows Compiler Dump Algorithm The Windows Compiler Dump Algorithm, as shown in Figure 6.4, is used when processing a Windows PE or PE+ executable. When Microsoft released its 64-bit version of Windows XP it extended the Windows PE format, called PE+, to support 64-bit executables. Therefore the Windows Trainer Dump Algorithm must first determine which type of file is being read,

load the file via the correct file reader, and then read the necessary attributes. This is shown in Figure 6.5.

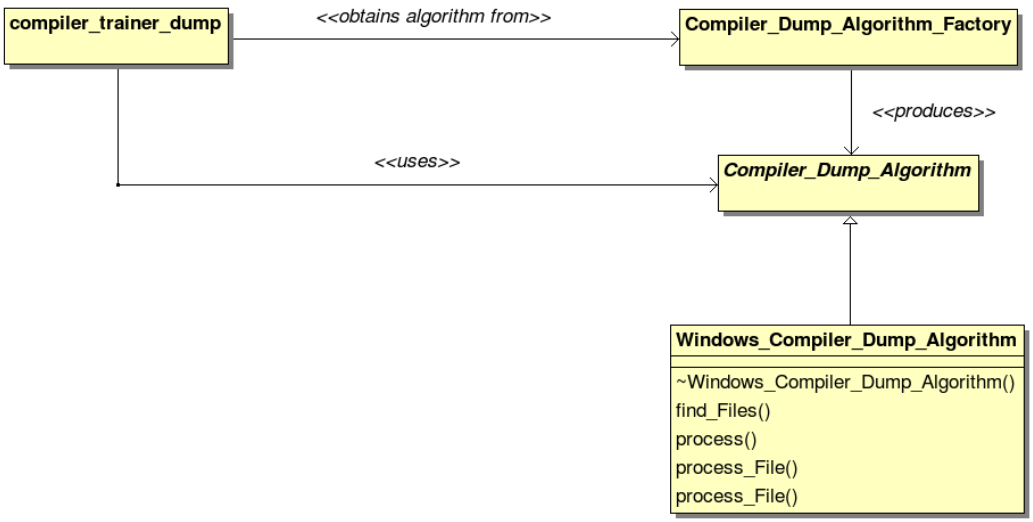


Figure 6.4: Windows Compiler Dump Algorithm class diagram

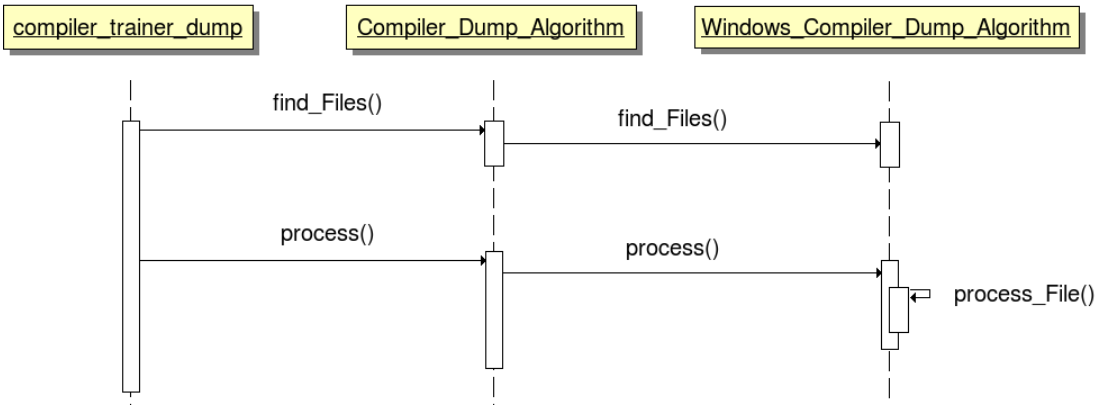


Figure 6.5: Windows Compiler Dump Algorithm sequence diagram for reading a Windows PE+ file

6.1.1.1.3 Java Compiler Dump Algorithm The Java Compiler Dump Algorithm, as shown in Figure 6.6, is used when processing a Java class file. Unlike the Linux ELF and Windows PE file format, there is no need to determine the kind of file that is being read. All the Java Compiler Dump Algorithm must do is load the file via

the Java file reader and then read the necessary attributes. This is shown in Figure 6.7. Another unique feature of this algorithm is the use of statistics about attributes in the file. When investigating the Java class file format it was noticed that certain kinds of objects in the Constant Pool table appeared more often for one compiler versus another. Therefore additional programming code was implemented to keep track of how many of each kind of Constant Pool table entry occurred. These operations are performed by the *collect_Constant_Pool_Stats* and *print_Constant_Pool_Stats* functions.

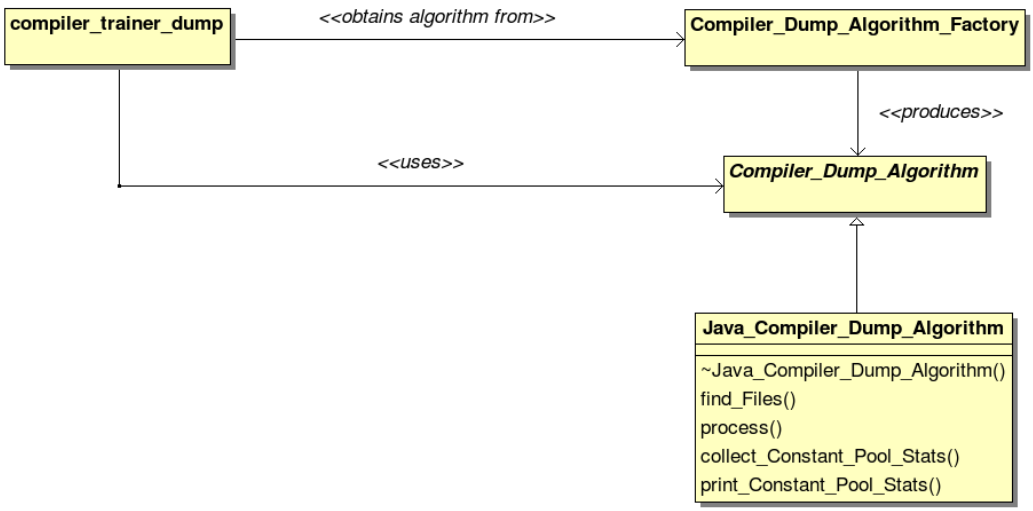


Figure 6.6: Java Compiler Dump Algorithm class diagram

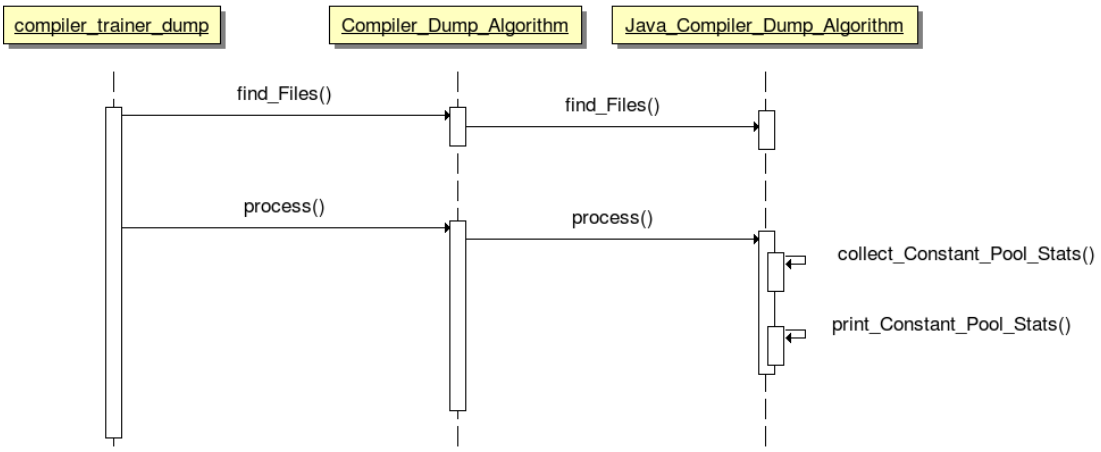


Figure 6.7: Java Compiler Dump Algorithm sequence diagram for reading a Java class file

6.2 Input Data

While a comma-delineated file makes it easy to switch between programs like Windows Excel and a text editor it does little in explaining what the information is and any expected fields. XML was chosen to represent the attributes collected from the input files since that it was designed [65] for:

1. XML shall be straightforwardly usable over the Internet.
2. XML shall support a wide variety of applications.
3. XML shall be compatible with SGML.
4. It shall be easy to write programs which process XML documents.
5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.
6. XML documents should be human-legible and reasonably clear.
7. The XML design should be prepared quickly.
8. The design of XML shall be formal and concise.
9. XML documents shall be easy to create.

10. Terseness in XML markup is of minimal importance.

It was important that the XML data should be easy to process, human readable, concise and easy to create. Since human developers are involved any ambiguities must be removed. XML has the option that requires files to adhere to a particular format. In this present design a schema is provided but not validated due to the limitation in the XML library used to parse the files. This is an important point in the future development of the compiler dump program.

6.2.1 Linux ELF XML

The input information used by the compiler classifier was obtained from a Compiler Dump program utilizes the Libreverse library, a reverse engineering library also written by the authors, to read the input ELF files. The input data was formatted in XML [6] in order to produce a easy parse input format to the classifier. The input data has the following XML schema [25] [61] [5] as shown in Listing 6.1.

Listing 6.1: XML schema for input ELF XML data

```
<?xml version=''1.0''?>
<xs:schema>
  <xs:element name=''data''>
    <xs:complexType>
      <xs:element name=''maximum'' type=''maximum_type''
                                    minOccurs="0"
                                    maxOccurs="1"/>
      <xs:sequence>
        <xs:element name=''file'' type=''file_type''
                                    minOccurs="1"
                                    maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name=''maximum_type''>
    <xs:sequence>
      <xs:element name=''value'' type=''xs:decimal''>
        <xs:complexType>
          <xs:attribute name=''key'' type=''xs:decimal''
                                      use=''required''/>
          <xs:attribute name=''float'' type=''xs:float''
                                      use=''required''/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name=''file_type''>
    <xs:sequence>
      <xs:element name=''compiler_id'' type=''xs:decimal''/>
      <xs:element name=''size'' type=''xs:decimal''/>
      <xs:element name=''entry_point_address'' type=''xs:decimal''/>
      <xs:element name=''section_headers_start'' type=''xs:decimal''/>
      <xs:element name=''program_header_count'' type=''xs:decimal''/>
      <xs:element name=''section_header_count'' type=''xs:decimal''/>
      <xs:element name=''section_header_string_table_index''
                  type=''xs:decimal''/>
      <xs:element name=''text_section_size'' type=''xs:decimal''/>
    </xs:sequence>
  </xs:complexType>
<xs:schema>
```

The description of each of the elements are:

- **compiler_id**: Unique id associated with the compiler that produce the file.

- **size**: File size
- **entry_point_address**: The address where program control is transferred.
- **section_header_start**: The file offset to the section header table.
- **program_header_count**: The number of program headers in the file.
- **section_header_count**: The number of section headers in the file.
- **section_header_string_table_index**: The index into the section header table associated with the string table.
- **text_section_size**: Size in bytes of the executable code section in memory.

6.2.2 Java Class XML

The input information used by the compiler classifier was obtained from a Compiler Dump program utilizes the Libreverse library, a reverse engineering library also written by the authors, to read the input Java Class files. The input data was formatted in XML [6] in order to produce a easy parse input format to the classifier. The input data has the following XML schema [25] [61] [5] as shown in Listing 6.2.

Listing 6.2: XML schema for Java XML data

```
<?xml version=''1.0''?>
<xs:schema>
  <xs:element name=''Data''>
    <xs:complexType>
      <xs:element name=''maximum'' type=''maximum_type''
                                    minOccurs="0"
                                    maxOccurs="1"/>
      <xs:sequence>
        <xs:element name=''file '' type=''file_type ''
                                    minOccurs="1"
                                    maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name=''maximum_type''>
    <xs:sequence>
      <xs:element name=''value'' type=''xs:decimal''>
        <xs:complexType>
          <xs:attribute name=''key'' type=''xs:decimal''
                                     use=''required''/>
          <xs:attribute name=''float '' type=''xs:float ''
                                     use=''required''/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name=''file_type''>
    <xs:sequence>
      <xs:element name=''version'' type=''xs:decimal''/>
      <xs:element name=''this_class_index '' type=''xs:decimal''/>
      <xs:element name=''super_class_index '' type=''xs:decimal''/>
      <xs:element name=''constant_pool_count '' type=''xs:decimal''/>
      <xs:element name=''constant_pool_info ''
                  type=''constant_pool_type ''
                  minOccurs=''12'' maxOccurs=''12''/>
      <xs:element name=''field_count '' type=''xs:decimal''/>
      <xs:element name=''method_count '' type=''xs:decimal''/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name=''constant_pool_type''>
    <xs:sequence>
      <xs:element name=''tag'' type=''xs:decimal''/>
      <xs:element name=''count'' type=''xs:decimal''/>
      <xs:element name=''ratio'' type=''xs:decimal''/>
    </xs:sequence>
  </xs:complexType>
<xs:schema>
```

The attributes that are collected are those that are independent of the java byte codes produced. The description of each of the elements are:

- **version**: This value is used to indicate what release level of Java this binary requires the JVM to support.
- **this_class_index**: The value of the this_class item must be a valid index into the constant_pool table. The constant_pool entry at that index must be a Constant Class Info structure representing the class or interface defined by this class file.[8]
- **super_class_index**: This is similar to the this_class defined above but the only difference is that this index refers to a parent class or the default Object class.
- **constant_pool_count**: Number of constant pool entries.
- **constant_pool_info (tag)**: Unique id for the constant pool entry.
- **constant_pool_info (count)**: Number of times the unique id was seen.
- **constant_pool_info (ratio)**: Ratio of the number times to the number of constant pool entries.
- **field_count**: Number of field_info structures in the class file.
- **method_count**: Number of method_info structures in the class file.

6.2.3 Windows PE XML

The input information used by the compiler classifier was obtained from a Compiler Dump program utilizes the Libreverse library, a reverse engineering library also written by the authors, to read the input Windows PE files. The input data was formatted in XML [6] in order to produce a easy parse input format to the classifier. The input data has the following XML schema [25] [61] [5] as shown in Listing 6.3. The description of the elements are:

- **compiler_id**: Unique id associated with the compiler that produce the file.

- **size**: File size
- **exe_header_address**: The address in the binary file where the PE header is located.
- **coff_section_header_count**: The number of section headers in the file.
- **pe_opt_code_size**: The size in bytes of the section that contains the executable instructions.
- **pe_opt_base_of_data**: The memory address where the data for the PE file starts.
- **pe_opt_entry_point**: The memory address where the first instruction to be executed is located.
- **pe_opt_image_size**: Size in bytes of the PE file in memory.

Listing 6.3: XML schema for input PE XML data

```
<?xml version=''1.0''?>
<xs:schema>
  <xs:element name=''data''>
    <xs:complexType>
      <xs:element name=''maximum'' type=''maximum_type''
                                    minOccurs="0"
                                    maxOccurs="1"/>
      <xs:sequence>
        <xs:element name=''file '' type=''file_type ''
                                   minOccurs="1"
                                   maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name=''maximum_type''>
    <xs:sequence>
      <xs:element name=''value'' type=''xs:decimal''>
        <xs:complexType>
          <xs:attribute name=''key'' type=''xs:decimal''
                                     use=''required''/>
          <xs:attribute name=''float '' type=''xs:float ''
                                       use=''required''/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name=''file_type ''>
    <xs:sequence>
      <xs:element name=''compiler_id '' type=''xs:decimal''/>
      <xs:element name=''size '' type=''xs:decimal''/>
      <xs:element name=''exe_header_address '' type=''xs:decimal''/>
      <xs:element name=''coff_section_header_count ''
                  type=''xs:decimal''/>
      <xs:element name=''pe_opt_code_size '' type=''xs:decimal''/>
      <xs:element name=''pe_opt_base_of_data '' type=''xs:decimal''/>
      <xs:element name=''pe_opt_entry_point '' type=''xs:decimal''/>
      <xs:element name=''pe_opt_image_size ''  type=''xs:decimal''/>
    </xs:sequence>
  </xs:complexType>
<xs:schema>
```

6.3 GRNN Optimizer

The grnn optimizer program was created to handle the offline classification of the compiler classification to determine the actual file parameters used and the optimal sigma value. The grnn optimizer has the follow command line flags:

- **directory**: Where the training data XML files are stored.
- **source type**: String name describing the type of test programs (e.g. Java class).
- **output**: Where the final training data file is stored.

The grnn optimizer program uses these command line flags to locate the appropriate algorithm to use, collect the training data XML files from the given directory, train and validate then finally store the condensed information in a XML file. Section 6.3.1 will describe each of the kind of algorithms available. Section 6.2 describes the XML schema associated with the training data.

6.3.1 Optimizer Algorithm

Each algorithm that is used to parse the XML training data must follow the interface laid out in Figure 6.8. The use of an abstract base class for the Optimizer Algorithm was necessary since it was determined to hide the kind of algorithm being used from the grnn optimizer program. It is unnecessary for the grnn optimizer program to know which algorithm it is interacting with when processing an XML file. This allows the design to support a wide variety of training data and allows scalability to future design changes.

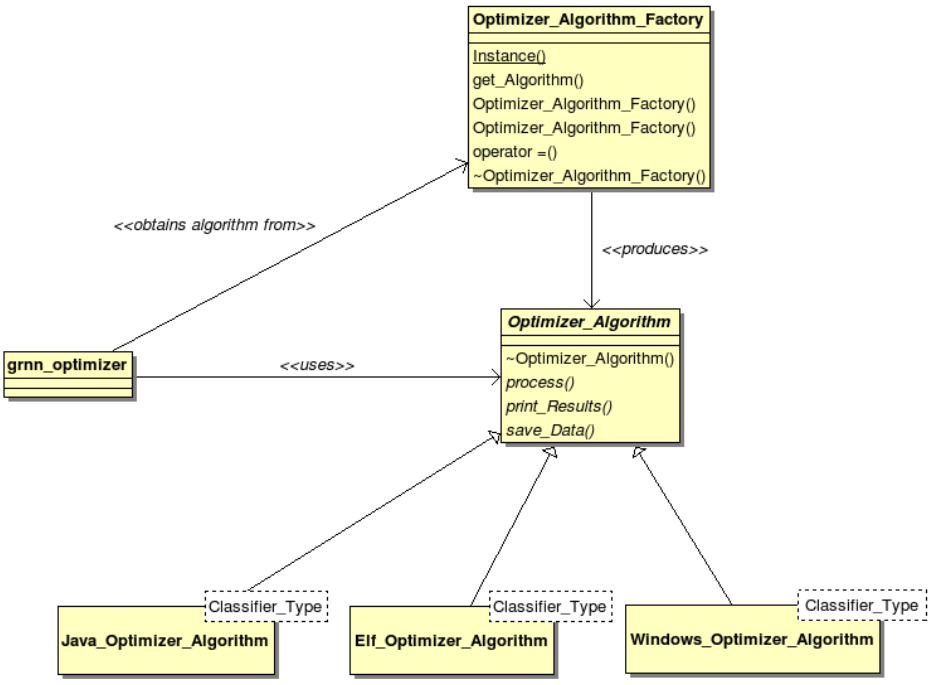


Figure 6.8: Optimizer Algorithm class diagram

The correct Optimizer Algorithm is found through the Optimizer Algorithm Factory's *get_Algorithm* function. The grnn optimizer program calls this function, obtains the correct algorithm, and then calls the algorithm's functions to perform its work. Each optimization algorithm was designed to accept the type of classifier to use as an template parameter. This design feature allows for the future expansion to additional classifier algorithms without duplication of functionality.

6.3.1.0.4 ELF Optimizer Algorithm The ELF Optimizer Algorithm, as shown in Figure 6.9, is used when processing a Linux ELF executable. The ELF Optimizer Algorithm parses in the input XML file name into a ELF Training Data object, runs the appropriate classifier and outputs the final XML data. This interaction is shown in Figure 6.10.

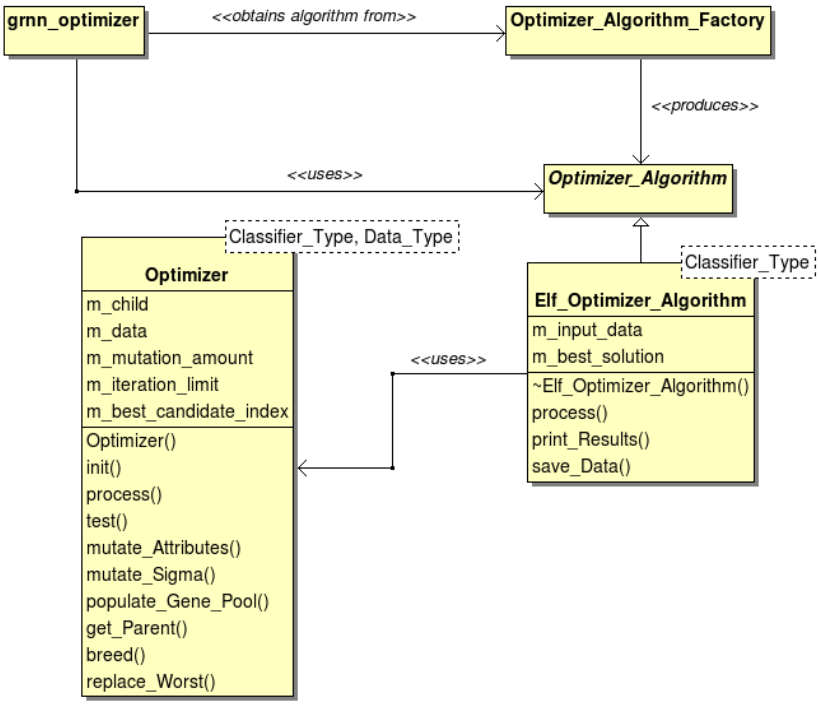


Figure 6.9: ELF Optimizer Algorithm class diagram

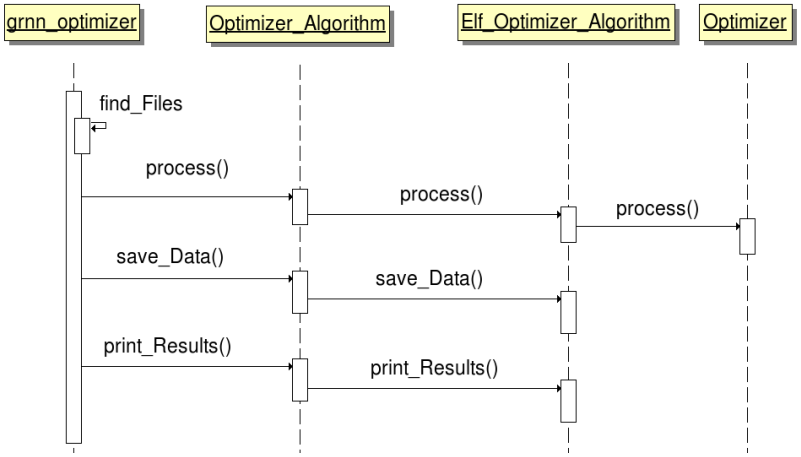


Figure 6.10: ELF Optimizer Algorithm sequence diagram

6.3.1.0.5 Java Optimizer Algorithm The Java Optimizer Algorithm, as shown in Figure 6.11, is used when processing a Java Class file. The Java Optimizer Algorithm

parses in the input XML file name into a Java Training Data object, runs the appropriate classifier and outputs the final XML data. This interaction is shown in Figure 6.12.

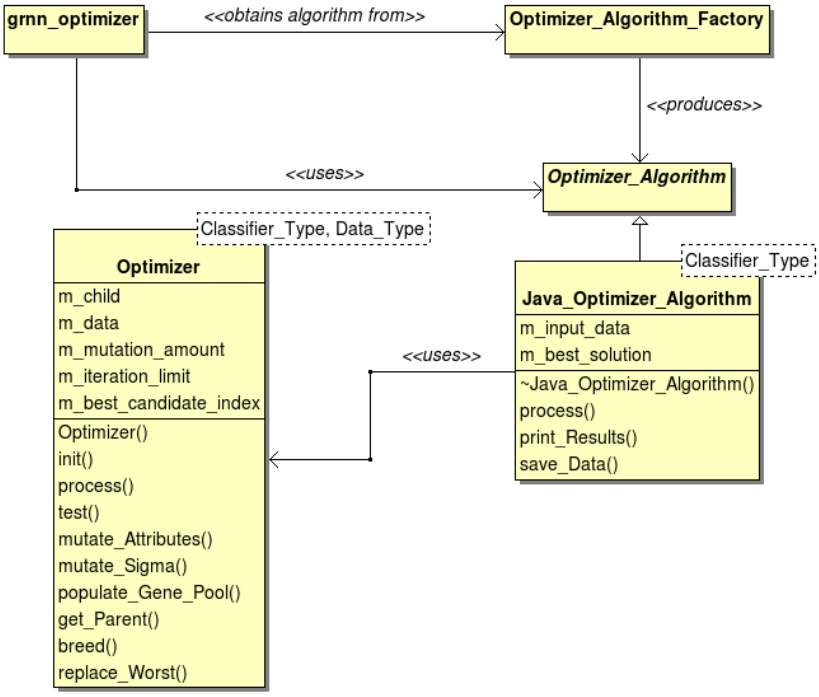


Figure 6.11: Java Optimizer Algorithm class diagram

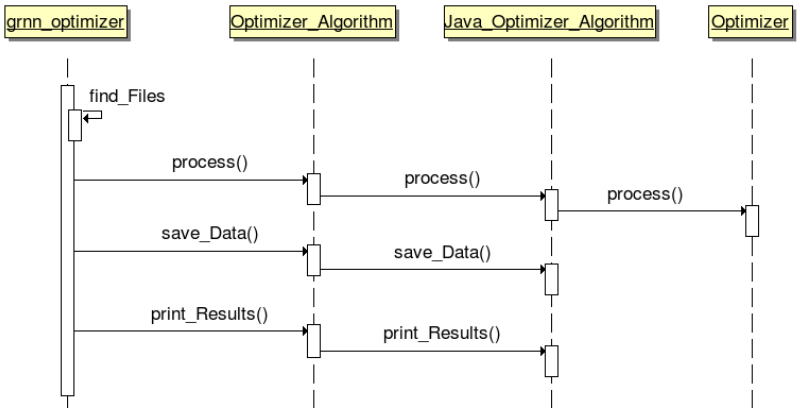


Figure 6.12: Java Optimizer Algorithm sequence diagram

6.3.1.0.6 Windows Optimizer Algorithm The Windows Optimizer Algorithm, as shown in Figure 6.13, is used when processing a Windows PE executable. The

Windows Optimizer Algorithm parses in the input XML file name into a Windows Training Data object, runs the appropriate classifier and outputs the final XML data. This interaction is shown in Figure 6.14.

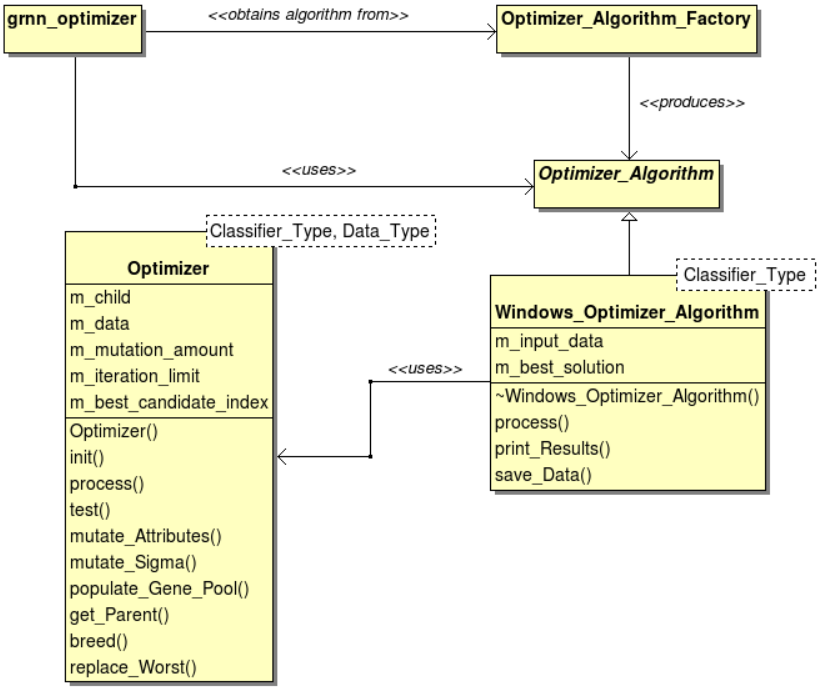


Figure 6.13: Windows Optimizer Algorithm class diagram

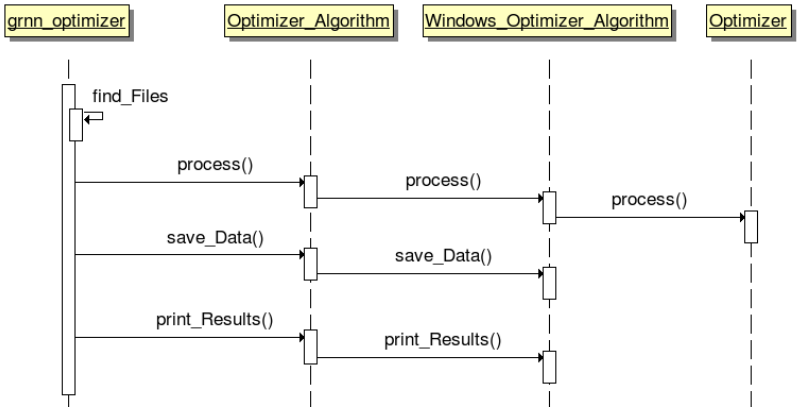


Figure 6.14: Windows Optimizer Algorithm sequence diagram

6.3.2 Input parser

Each of the Optimization algorithms described in Section 6.3.1.0.4, 6.3.1.0.5 and 6.3.1.0.6 rely on the IO class to parse the stored XML data. The XML data is the result of running the grnn optimizer program on the collected training data. The parsing of the stored XML data is covered in Section 6.3.2.1 and the data structure returned from the parser is described in Section 6.3.2.3.

6.3.2.1 IO

The IO class is use during the on-line and offline phases of compiler classification. The first 'get_Data' function that takes a list of input XML data files is used during the offline phase. The pseudo code of this function is covered in Algorithm 8.

Algorithm 8 IO - get_Data (offline phase)

```
1: procedure GET_DATA(F)
2:     Create an instance of the correct parser
3:     for f ∈ F do
4:         if f does NOT exist then
5:             Throw an exception
6:         end if
7:         Parse the Training Set from f using the parser
8:     end for
9:     Normalize the Training Set
10:    Separate the Training Set into 60% training, 20% test and 20% validation data
11:    Save the maximum value for each column of the Training Set.
12:    return sorted Training Set.
13: end procedure
```

The algorithm takes in a Training Set containing N different entries and prepares them for use in the optimization operation. The normalization of data is required when using data with a neural network [20] because normalization enables comparison between the various file attributes. The normalized data is separate into the values use to learning from (training data), check the resulting learner (test data) and verify its correct (validation data). All the information is now ready for the optimizer and GRNN to learn if there is any means to distinguish the various classes of input from each other.

The second 'get_Data' function is used for reading the stored normalized training data that was the result of the grnn optimizer. This information is utilized by the Compiler Classifier Component described in Section 9.4.11. There is no need to normalize and separate the data again like done in the first 'get_Data' function. The data is converted from its XML format into the appropriate Training Set object for use by the Compiler Classifier Component. Algorithm 9 describes the operations of this function.

Algorithm 9 IO - get_Data (on-line phase)

```
1:  procedure GET_DATA(f)
2:      if f does NOT exist then
3:          Throw an exception
4:      end if
5:      Create an instance of the correct parser
6:      Parse the training set from f using the parser
7:      Create new Training Set object
8:      Add input data to the Training Set object
9:      Add the maximum values to the Training Set object
10:     return sorted Training Set object.
11: end procedure
```

6.3.2.2 Training Set

The training set is a data container for storing the training, test and verification information in training data objects. Each Training Set is specialized by the kind of training data objects stored in it through template specialization. Various methods are available to allow iteration through the various training data object's values during the optimization and classification operations.

6.3.2.3 Training Data

The training data is a data container that stores the value for one kind of training information (e.g. test data for Linux ELF) specialized by a template parameter. It is a standard template library map that is designed to use the attribute id as the key to set or retrieve information. Each optimization algorithm for a particular file type, e.g. ELF_Optimizer_Algorithm, uses predefined attributes in these specialized training data

classes to access values stored in the training data. The following sections describe each of the specialized training data classes. Each of the specialized Training Data classes provide predefined constants, "ATTRIBUTE_COUNT" and "CLASSIFIER_TARGET"', that are used by the Optimizer and GRNN classes during the optimization process. These values allow for the Optimizer to know how many attributes it is consider and which target the data is assigned to without knowing anything about the data. By design the "CLASSIFIER_TARGET" is always the first attribute in each training data object. Therefore "CLASSIFIER_TARGET" is set to 0. Each class also has an enumeration which lists the unique id for each attribute. The names and unique ids are used are the result of the expert determining the file attributes that could potentially be used for classification. How many file attributes used in each training data object is set in the "ATTRIBUTE_COUNT".

6.3.2.3.1 ELF Training Data The ELF Training Data class is used to specialize the training data class for storing values retrieved from the input XML data. It has its "ATTRIBUTE_COUNT" set to 8 and lists the values in its Attributes enumeration in Table 6.1.

Name	ID
ATTRIBUTE_TARGET_ID	0
ATTRIBUTE_FILESIZE	1
ATTRIBUTE_ENTRY_POINT_ADDRESS	2
ATTRIBUTE_SECTION_HEADERS_START	3
ATTRIBUTE_PROGRAM_HEADER_COUNT	4
ATTRIBUTE_SECTION_HEADER_COUNT	5
ATTRIBUTE_SECTION_HEADER_STRING_TABLE_INDEX	6
ATTRIBUTE_TEXT_SECTION_SIZE	7

Table 6.1: ELF Training Data - Attributes enumeration

6.3.2.3.2 Java Training Data The Java Training Data class is used to specialize the training data class for storing values retrieved from the input XML data. It has its "ATTRIBUTE_COUNT" set to 32 and lists the values in its Attributes enumeration in Table 6.2.

Name	ID
ATTRIBUTE_TARGET_ID	0
ATTRIBUTE_FILESIZE	1
ATTRIBUTE_THIS_INDEX	2
ATTRIBUTE_SUPER_INDEX	3
ATTRIBUTE_VERSION	4
ATTRIBUTE_CONSTANT_POOL_COUNT	5
ATTRIBUTE_CONSTANT_UTF8_COUNT	6
ATTRIBUTE_CONSTANT_UTF8_RATIO	7
ATTRIBUTE_CONSTANT_RESERVED_COUNT	8
ATTRIBUTE_CONSTANT_RESERVED_RATIO	9
ATTRIBUTE_CONSTANT_INTEGER_COUNT	10
ATTRIBUTE_CONSTANT_INTEGER_RATIO	11
ATTRIBUTE_CONSTANT_FLOAT_COUNT	12
ATTRIBUTE_CONSTANT_FLOAT_RATIO	13
ATTRIBUTE_CONSTANT_LONG_COUNT	14
ATTRIBUTE_CONSTANT_LONG_RATIO	15
ATTRIBUTE_CONSTANT_DOUBLE_COUNT	16
ATTRIBUTE_CONSTANT_DOUBLE_RATIO	17
ATTRIBUTE_CONSTANT_CLASS_COUNT	18
ATTRIBUTE_CONSTANT_CLASS_RATIO	19
ATTRIBUTE_CONSTANT_STRING_COUNT	20
ATTRIBUTE_CONSTANT_STRING_RATIO	21
ATTRIBUTE_CONSTANT_FIELDREF_COUNT	22
ATTRIBUTE_CONSTANT_FIELDREF_RATIO	23
ATTRIBUTE_CONSTANT_METHODREF_COUNT	24
ATTRIBUTE_CONSTANT_METHODREF_RATIO	25
ATTRIBUTE_CONSTANT_INTERFACE_METHODREF_COUNT	26
ATTRIBUTE_CONSTANT_INTERFACE_METHODREF_RATIO	27
ATTRIBUTE_CONSTANT_NAME_AND_TYPE_COUNT	28
ATTRIBUTE_CONSTANT_NAME_AND_TYPE_RATIO	29
ATTRIBUTE_FIELD_COUNT	30
ATTRIBUTE_METHOD_COUNT	31

Table 6.2: Java Training Data - Attributes enumeration

6.3.2.3.3 Windows Training Data The Windows Training Data class is used to specialize the training data class for storing values retrieved from the input XML data. It has its "ATTRIBUTE_COUNT" set to 8 and lists the values in its Attributes enumeration in Table 6.3.

Name	ID
ATTRIBUTE_TARGET_ID	0
ATTRIBUTE_FILESIZE	1
ATTRIBUTE_EXE_HEADER_ADDRESS	2
ATTRIBUTE_COFF_SECTION_COUNT	3
ATTRIBUTE_PE_OPT_CODE_SIZE	4
ATTRIBUTE_PE_OPT_BASE_OF_DATA	5
ATTRIBUTE_PE_OPT_ENTRY_POINT	6
ATTRIBUTE_PE_OPT_IMAGE_SIZE	7

Table 6.3: Windows Training Data - Attributes enumeration

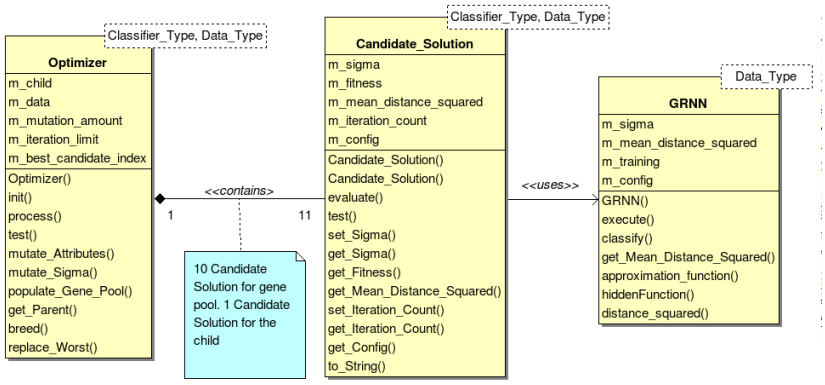


Figure 6.15: Optimizer class diagram

6.3.2.4 Training Data Parser

Each input training data is stored with a different XML schema as shown in Section 6.2.2 of Java Class, Section 6.2.1 for Linux ELF and Section 6.2.3 for Windows PE. An XML parser is used to convert the data in XML format to a Training Set object.

6.3.3 Optimizer

The Optimizer object, shown in Figure 6.15, is responsible for discovering the best possible solution for a given input data, sigma and mutation value. It can be specialized with a classifier and data type to be used thereby allowing for replacing the types of classifier algorithm used and the input data type. The function of the Optimizer object utilized by the Optimizer algorithms is the 'process' as described in Algorithm 10 and shown in Figure 6.16.

Algorithm 10 Optimizer - process function pseudo code

```
1:  procedure PROCESS
2:      Populate the gene pool with 10 Candidate Solution objects
3:      for i ← 0 to iteration limit do
4:          Evaluate each Candidate Solution in gene pool
5:          Get the first randomly selected parent from the gene pool
6:          Get the second randomly selected parent from the gene pool
7:          Breed the two parents to produce a child Candidate Solution
8:          Mutate the child Candidate Solution attributes
9:          Calculate mutation chance between 1 to 10
10:         if mutation chance is 1 then
11:             Mutate Sigma value
12:         end if
13:         Evaluate child Candidate Solution
14:         Set the iteration count in the child Candidate Solution
15:         Replace the worst Candidate Solution found in the gene pool with the child
16:     end for
17:     Save the best Candidate Solution in the gene pool
18: end procedure
```

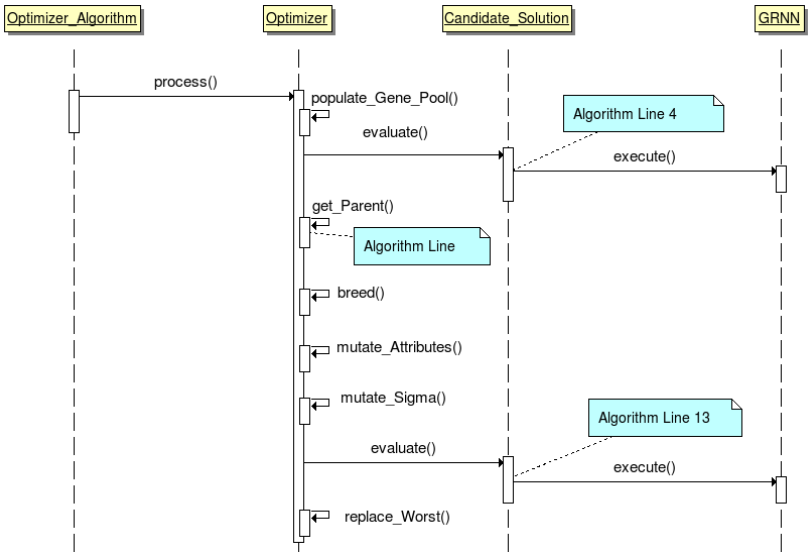


Figure 6.16: Optimizer sequence diagram

6.3.4 Candidate Solution

The Candidate Solution, as shown in Figure 6.15, is a data structure used to keep track of all the possible iterations used in the Optimizer gene pool. In the present implementation of Libreverse the Candidate Solution is instantiated with the GRNN classifier and the appropriate data structure for the optimizer algorithm.

6.3.5 GRNN

The GRNN, as shown in Figure 6.15, implements the principles described in Section 3.2 for use in performing the classification during the off-line and on-line phases.

Chapter 7

API

Applications built against Libreverse work through the API interface directly. The interface controls the operation of the Libreverse library.

7.1 Execute

This function, as shown in Figure 7.1, coordinates all the specific operations of the reverse engineering of the target file. The idea behind this function was that most users of Libreverse will not have the detailed knowledge of what is required to be done. Therefore this function automates all the operations for the user.

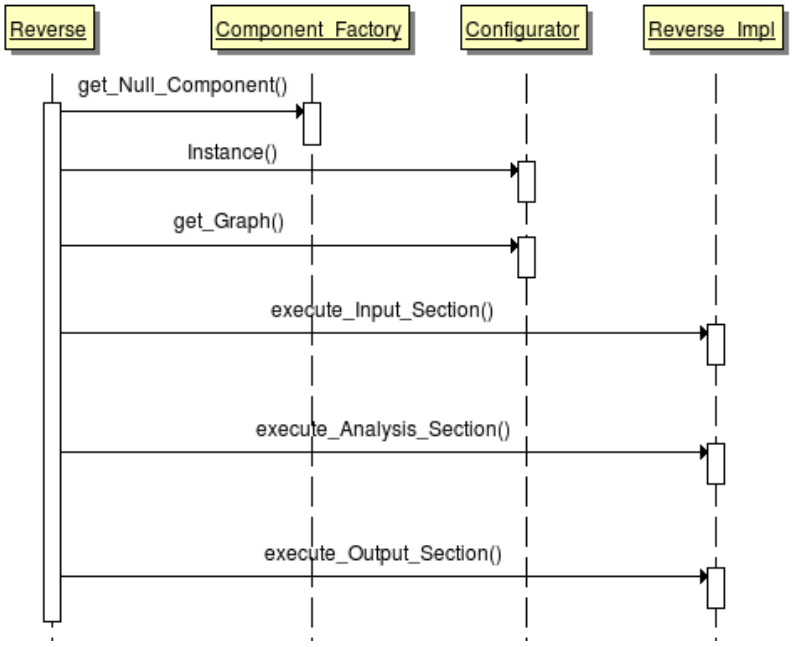


Figure 7.1: Libreverse API 'execute' function

Parameter	Description
target_file	File to be processed by Libreverse.
input_type	File type of the input (e.g. binary).
output_type	Desired output (e.g. C++).
trace_level	Tracing level for recording debugging information.
trace_mask	Bitwise OR value that represents where debugging information is recorded.

Table 7.1: Parameters of Execute function

7.1.1 Steps of Processing

The Reverse class is the location that handles the actual processing of the target file given the input and output type. The execute function takes the parameters, as shown in Table 7.1, and performs as shown in Algorithm 11.

Algorithm 11 Reverse - execute

1: **procedure** EXECUTE($target$,$type_{input}$,$type_{output}$, $debug_{level}$, $debug_{mask}$)
2: Determine if target file exists.
3: Check desired input and output type are valid.
4: Prepare initial source component.
5: Prepare initial data source that contains the target file name.
6: Get the Component Graph associated with the input and output type
7: Execute the input section
8: Execute the analysis section
9: Execute the output section
10: **end procedure**

The first two steps are performed to ensure everything provided by the application calling 'execute' is correct and available before beginning the processing of the target file.

In the third and fourth step, the input Data Source, described later in Chapter 11, is prepared with the given target file name for use with the source component. A Null Component, described later in Section 9.4.8, acts as the source component providing information to the first component in the Component Graph, described in Chapter 10.

In the fifth step, the Component Graph to be used is obtained from Configurator, described in Section 8.1, using the input and output type. The Component Graphs are predefined arrangements of components that represent the best practices of reverse

engineering discovered by the research community. How the graph is obtained is defined in Chapter 8 and constructed in Chapter 10.

The sixth through eighth steps process the target file using the Component Graph. This operation is handled by the Reverse_Impl class. In order to hide the internal operations of the processing stages the Reverse_Impl class was designed to offer three functions. The first function, execute_Input_Section, processes the Input section of the Component Graph. The second function, execute_Analysis_Section, processes the Analysis section of the Component Graph. Finally execute_Output_Section processes the Output section of the Component Graph. Information is passed between the sections by the Execute function.

Each of the processing functions in Reverse_Impl class follows the same algorithm as shown in Algorithm 12.

Algorithm 12 Reverse_Impl - execute

```
1: procedure EXECUTE
2:     Obtain the Component Graph for that part (e.g. input ).
3:     Print the Component Graph in graphviz dot file format for use in documentation.
4:     Initialize Component Graph with the source component
5:     Process each component in the Component Graph
6:     Retrieve the results from the last component.
7: end procedure
```

The idea behind splitting up the steps of reverse engineering was inspired by the way Cifuentes [14] and the Boomerang [35] project performed the major steps of reverse engineering. Various uses of reverse engineering have similar input (e.g. Windows PE executable) but have different analysis and output operations.

Chapter 8

Configuration

Libreverse is dynamically configured at run time based on the kind of input and output the caller to the API *execute* function. The controller of Libreverse's configuration is the Configurator class covered in Section 8.1. It provides a variety of information to various parts of Libreverse. Figure 8.1 shows the class diagram of the Configurator.

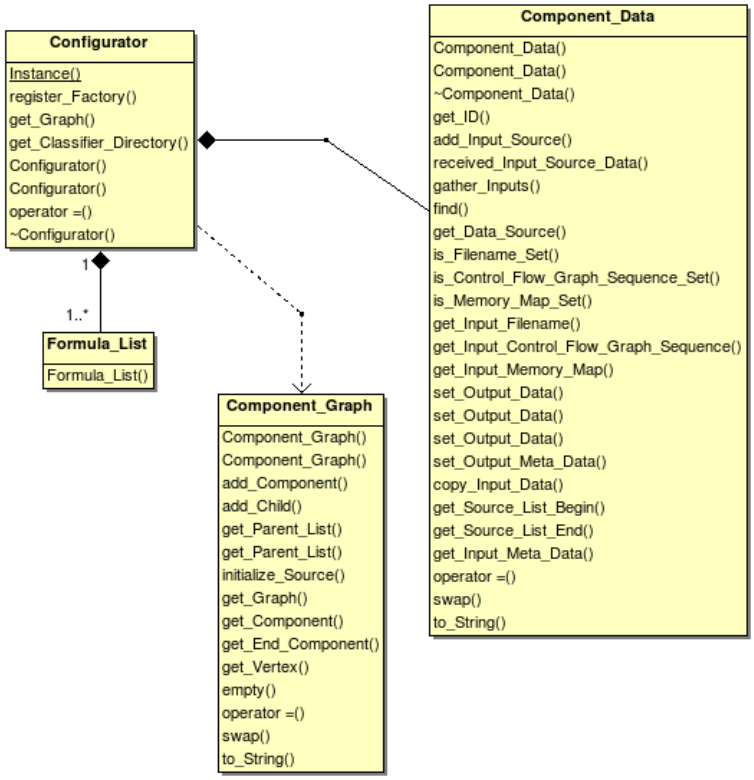


Figure 8.1: Configurator Class Diagram

8.1 Configurator

The Configurator is a singleton object that provides the Component Graphs for processing a target binary file and sets up the Data Source factory for creating Data

Source objects used to pass information. The reason behind having the Configurator being a singleton was the fact that there should only be one source of information. Therefore the Singleton pattern was employed in this case.

The first responsibility of the Configurator is to create three Component Graphs composed of Components prearranged by the associated Formula File. The input and output types given to the *get_Graph* function retrieves the appropriate master formula file. Each master formula file contains a text string for the name of the formula file for each of the sections (Input, Analysis, and Output). This is seen in Figure 8.2. The pseudo code for *get_Graph* is shown in Algorithm 13.

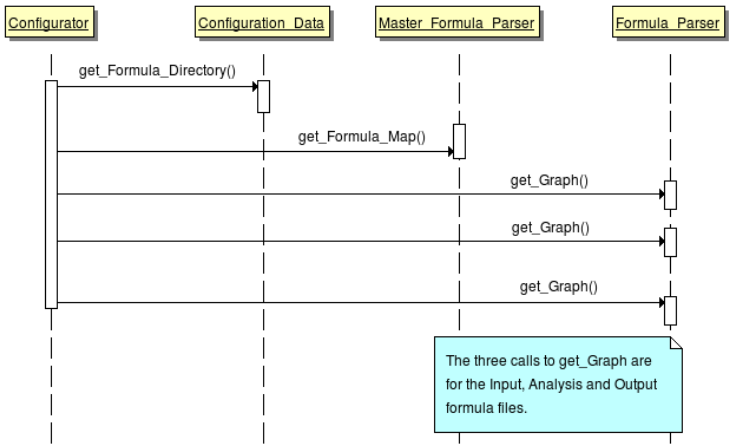


Figure 8.2: Configurator *get_Graph* sequence diagram

The second responsibility of the Configurator is to setup the Data Source Factory for creating Data Source objects are created for transferring information from one component to the next. At this time only the memory passing Data Source is produced to transfer information between components. In the future additional Data Sources will be supported as shown in Section 15.2.6.

8.2 Master Formula File

The purpose of the master formula file is to show the three formula files that compose the best practice solution for a specific problem. An entry in the master

Algorithm 13 Configurator - get_Graph

```
1:  procedure GET_GRAPH(IT,OT)
2:      if Master Formula Map is not set then
3:          Get formula directory from Configuration Data.
4:          Read the Master Formula Map
5:      end if
6:      Search for correct Formula Map in the Master Formula Map using IT, OT
7:      if Formula Map is not found then
8:          throw an exception
9:      end if
10:     Parse the input Component Graph with a Formula Parser
11:     Parse the analysis Component Graph with a Formula Parser
12:     Parse the output Component Graph with a Formula Parser
13:     return resulting Component Graphs
14: end procedure
```

formula file is composed of a *input* and *output* value that represent the key pair for the solution. The *input_formula*, *analysis_formula* and *output_formula* store the string name of the formula file for respective for the Input, Analysis and Output Component Graphs. A high level view of how the formula files work together are seen in Figure 8.3. Figure 8.1 describes the XML schema for the master formula file.

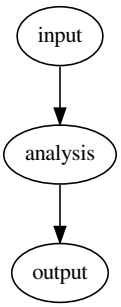


Figure 8.3: High-level view of set of Component Graphs

Listing 8.1: XML schema for master formula file

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="formula_list">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="formula" type="FormulaType" minOccurs="1"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="FormulaType">
    <xs:sequence>
      <xs:element name="input" type="InputType"/>
      <xs:element name="output" type="OutputType"/>
      <xs:element name="input_formula" type="FormulaName"/>
      <xs:element name="analysis_formula" type="FormulaName"/>
      <xs:element name="output_formula" type="FormulaName"/>
    </xs:sequence>
  </xs:complexType>
  <xs:simpleType name="FormulaName">
    <xs:restriction base="xs:string">
      <xs:pattern
        value="[a-zA-z0-9]([a-zA-z0-9]|_)*[a-zA-z0-9]\.fm"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="InputType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="binary"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="OutputType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="C++"/>
      <xs:enumeration value="C"/>
      <xs:enumeration value="Java"/>
      <xs:enumeration value="UML"/>
    </xs:restriction>
  </xs:simpleType>
</xs:schema>
```

8.3 Formula File

The Formula File represents the best practice for a particular part, e.g. input phase, of the reverse engineering process. The file represents the arrangement of components in a Component Graph stored in XML format using schema shown in Listing 8.2. Each

Formula file has a series of *node* elements that make up the graph. Each *node* element has a list of predecessors providing it data and meta information plus a *name* element describing the node's name and unique id.

Each node must follow a set of rules with the following variables.

- Child node: If a *node* has predecessors then the predecessor nodes must be exist already in Component Graph therefore no parent node can reference a unknown child. This rule prevents any loops from occurring in the resulting Component Graph.
- A parent node will have a lower unique id than any of its children.
- No duplicates. Each *node* must provide a unique *id* attribute for the Component Graph being constructed.

Listing 8.2: XML schema for formula file

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="formula_map">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="node"
                    type="NodeType"
                    minOccurs="1"
                    maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
    <xs:key name="nodeId">
      <xs:selector xpath="./node/id"/>
      <xs:field xpath="@id"/>
    </xs:key>
    <xs:key name="predNodeRef">
      <xs:selector xpath="./node/predecessors"/>
      <xs:field xpath="@idref"/>
    </xs:key>
  </xs:element>
  <xs:complexType name="NodeType">
    <xs:sequence>
      <xs:element name="predecessors" type="NodeReference"/>
      <xs:element name="name" type="xs:string">
        <xs:complexType>
          <xs:attribute name="id" type="xs:NCName" use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="NodeReference">
    <xs:sequence>
      <xs:element name="node_ref" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:attribute name="idref" type="xs:NCName" use="required"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Chapter 9

Components

9.1 Design Principles

A component is defined as an object with two modes of operation (source and Working) that is loosely coupled from other components performing a single primitive operation. Previous reverse engineering solutions [35] [14] [41] utilized functions in the programming language to provide the primitive operations. The problem with these approaches is that they do not allow the user to rearrange these primitive operations for new problems without altering the programming source code.

The list of design principles are listed here and describe in further detail below.

- Two mode operation (source vs working)
- Single primitive operation
- Loosely coupled
- Type of source of input data and destination of output data hidden

9.1.1 Two Mode Operation

When a component is in the working mode it is designed to operate as a part of a component graph, defined later in Chapter 10, to perform its specific action. Each child component takes its input data from their parent components through the data source, defined later in Chapter 11. The input data is retrieved from a list of data sources contained in the graph visitor defined later in Chapter 10. If all the necessary sources of input data are provided the component when in the working mode it then performs its specific operation. The results are placed into a data source and entered into the list of data sources for use by other components.

When a component is the head node of the component graph it must have its state set to source mode. When a component is in the source mode its normal operation is not performed. The role of a component in this state is to provide its data contained in a data source to child components in the component graph. For example, the file name of a target binary executable to be analysis can be provided by a Null component set in the source mode. How the graph visitor interacts with the components is described in Chapter 10.

9.1.2 Single Primitive Operation

Any complex reverse engineering technique is often composed of many primitive operations. These primitive operations are often reused across many different techniques. In order to capture this principle each of the components described in this chapter are design to only perform a single primitive operation. Couple this design principle with the loosely coupled nature and the data sources then a component can be reused multiple times in the same component graph without any knowledge to the components before or after it.

9.1.3 Loosely Coupled

Each component is designed to accept a visitor interacting with it and a set of data sources. There are no hard links allowed between various components since how a generic reverse engineering architecture will be used cannot be known before hand. Therefore each component must be designed to be moved into various places in a component graph thereby giving it nearly infinite flexibility to handle a vast amount of unknown problems.

9.1.4 Hidden data sources

Since a design goal was to maintain a loosely coupled connectivity between components there had to be a means by which the data could be passed. Each data source contains all the information provided by a parent component but without the child

component needing to know exactly who is its parent. This is important since hard wiring components together severely limits the range of problems this architecture can solve. Each data source not only hides the previous component from the next but it also has the ability to transmit the information through different means. This is further described in Chapter 11.

9.2 Component Class Diagram

The component class is a purely virtual class that represents the interface, shown in Table 9.1, that hides the internal workings from other parts of Libreverse. The component base class implements the features of the component interface and is the parent class that developers must inherit from when making a new component for Libreverse. The component base class also implements the component actor interface described in Section 9.3. The component base class has a member variable for a Component State object which implements the two-mode operation. The overall structure of a component is shown in Figure 9.1.

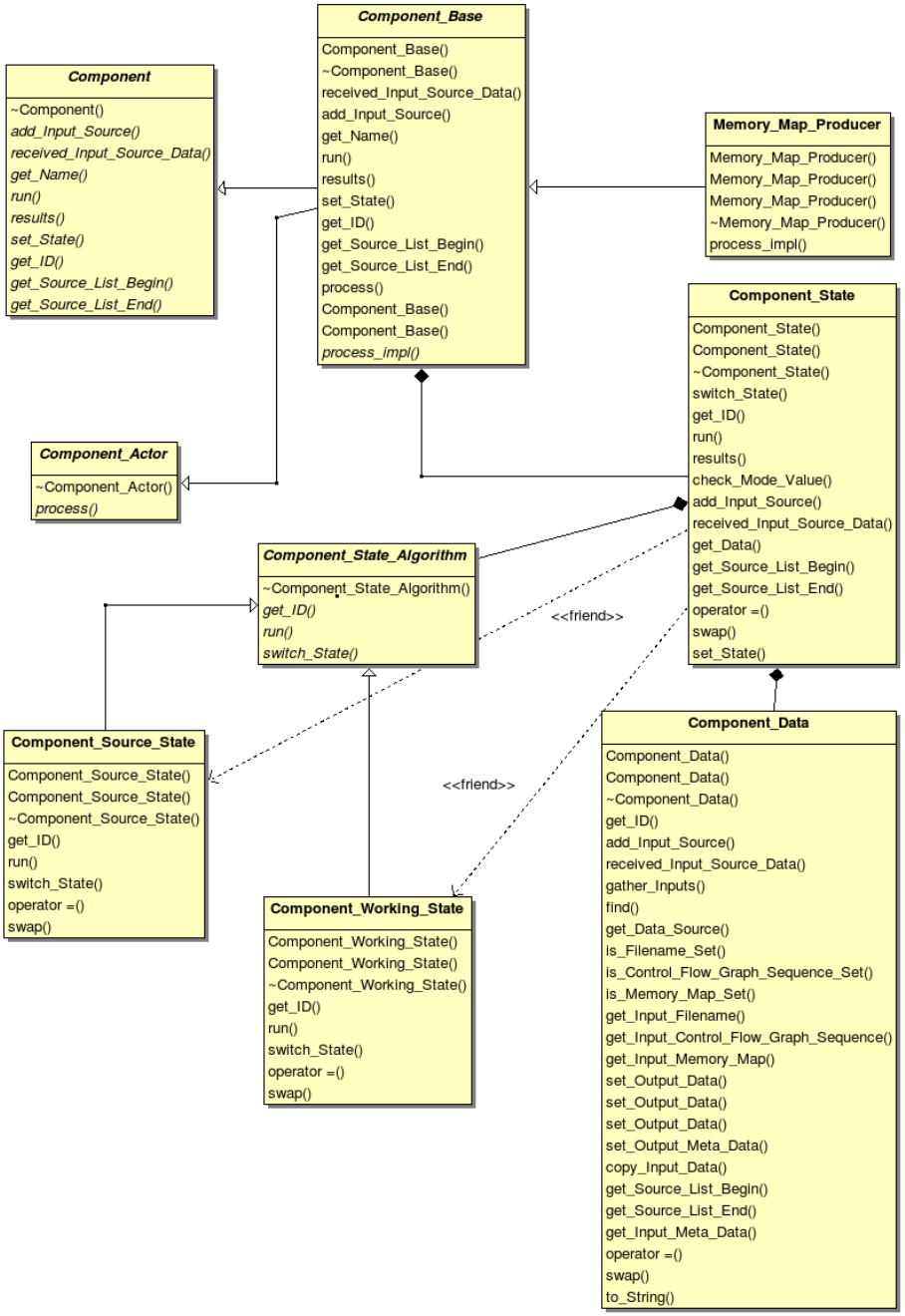


Figure 9.1: Component Class Diagram

Function	Description
Add Input Source	This is used during the creation of the Component Graph. Each parent component providing data to a child component must give its id to the child. The child uses this id to check to see if it has all its parent components have completed their operation before running its analysis.
Received Input Source Data	This is used during the actual processing of the component graph. The parent component, identified by id, has completed its analysis and has recorded its results into the data map type in the graph visitor. The child component marks that it has received input from the parent component.
Get Name	This is used for obtaining the actual name of the component when printing information for debugging or producing a graphviz file of the component graph
Run	This is used by the graph visitor when it visits the node that contains a component. This starts the primitive operation performed by the component.
Results	Returns the data source object that contains the output data. This is called by the graph visitor to store the results and notify the child components of the completion of the parent components operation.
Set State	This is functions will set the mode of the component. It is mainly used to set a component to the SOURCE_MODE when the Reverse_Impl class sets the component acting as a source of information to a component Graph.
Get Id	Return the Unique ID for the component.
Get Source List Begin	The input token list contains the list of parent components that supply the child component with data. If the data is ready the Boolean value is true otherwise it is false.
Get Source List End	The input token list contains the list of parent components that supply the child component with data. If the data is ready the Boolean value is true otherwise it is false.

Table 9.1: Component Interface

9.3 Component Actor Interface

The component actor interface is the mechanism by which the two mode operation of the component object is implemented. The component base class inherits from the

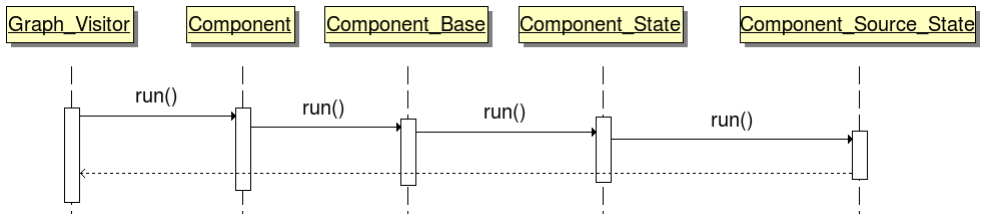


Figure 9.2: Component execution with its state set to SOURCE_MODE

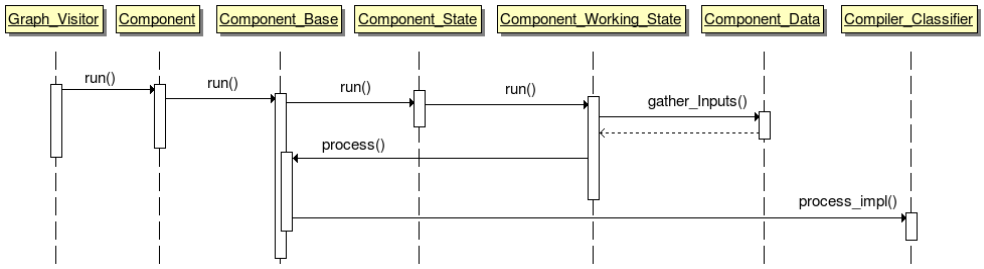


Figure 9.3: Component execution with its state set to WORKING_MODE

component actor to implement the 'process' function that will be called by the appropriate component state class (Component_Source_State or Component_Working_State) depending on which is active in the Component at that time. If the component is in the SOURCE_MODE then the 'run' function does nothing since the component base class 'process' function is never called. This is shown in Figure 9.2. Otherwise if the component is in WORKING_MODE then the 'run' function will call 'process' function on the component base so that the primitive operation will be performed. This is shown in Figure 9.3. The component base class calls the 'process_impl' function in the child class to execute the primitive operation.

9.4 Input Components

This section contains a list and description of the components provided as a part of the Libreverse architecture for use in the "input" formula files.

9.4.1 Architecture Type Detector

The Architecture Type Detector Component determines the CPU architecture that is supported by this binary executable. For most architectures there is some form of

identifier that the operating system loader uses when determining if it can execute an binary executable. There is little incentive for an attacker to alter the id in the binary executable since they want the target program to execute. Algorithm 14 shows the pseudo code for this component.

Algorithm 14 Architecture Type Detector - process_impl

```
1: procedure PROCESS_IMPL
2:     Get the component data object from the input data source.
3:     if target file name exists then
4:         Get the target file name from the component data object
5:     else
6:         Throw an exception
7:     end if
8:     Get the file type from the meta object found in the component data object.
9:     Get the correct Arch_Type_Detector_Algorithm object from the based on the file
   type from the Arch_Type_Detector_Algorithm_Factory.
10:    Execute the algorithm
11:    Store the returned Arch_Type_Meta object in the output data source.
12: end procedure
```

9.4.2 Code Section Detector

The Code Section Detector determines the beginning address in memory of the executable code of a binary executable and size of memory used. The Code Section Detector uses the input file name obtained from the data source and the appropriate file reader produced by the File Reader Factory to read the necessary field. This information is appended to the output data source as meta information. Algorithm 15 shows the pseudo code for this component.

Algorithm 15 Code Section Detector - process_impl

```
1: procedure PROCESS_IMPL
2:     Get the component data object from the input data source.
3:     if target file name exists then
4:         Get the target file name from the component data object
5:     else
6:         Throw an exception
7:     end if
8:     Get the file type from the meta object found in the component data object.
9:     Get the correct Code_Section_Detector_Algorithm object from the based on the
   file type from the Code_Section_Detector_Algorithm_Factory.
10:    Execute the algorithm
11:    Store the returned meta object in the output data source.
12: end procedure
```

Each of the Code Section Detector algorithms uses its associated File Reader to obtain the code section information. Windows PE, Linux ELF and Mac OS.X MachO all allocate memory where the executable instructions are then placed by the operating system loader. Java class files have Code Attribute sections that are part of each method. Each class file is loaded by the JVM as it is required. There is no memory allocated in a similar way compared to the other formats. Therefore there is no real value that can be returned for this nor of the section size.

9.4.3 Data Section Detector

The Data Section Detector determines the beginning address in memory of the program data in the binary executable and size of memory used. The Data Section Detector uses the input file name obtained from the data source and the appropriate file reader produced by the file reader factory to read the necessary field. This information is appended to the output data source as Meta information. Algorithm 16 shows the pseudo code for this component.

Algorithm 16 Data Section Detector - process_impl

```
1: procedure PROCESS_IMPL
2:     Get the component data object from the input data source.
3:     if target file name exists then
4:         Get the target file name from the component data object
5:     else
6:         Throw an exception
7:     end if
8:     Get the file type from the meta object found in the component data object.
9:     Get the correct Data_Section_Detector_Algorithm object from the based on the
   file type from the Data_Section_Detector_Algorithm_Factory.
10:    Execute the algorithm
11:    Store the returned meta object in the output data source.
12: end procedure
```

Each of the Data Section Detector algorithms uses its associated File Reader to obtain the data section information. Windows PE, Linux ELF and Mac OS.X MachO all allocate memory where the application data is placed by the operating system loader. Java class files have Code Attribute sections that are part of each method. Each class file is loaded by the JVM as it is required. There is no memory allocated in a similar way compared to the other formats. Therefore there is no real value that can be returned for this nor of the section size.

9.4.4 Entry Point Detector

The Entry Point Detector determines the beginning address in memory of the first executable instruction is located in the binary executable. The Entry Point Detector uses the input file name obtained from the data source and the appropriate file reader produced by the file reader factory to read the necessary fields. This information is appended to the output data source as meta information. Algorithm 17 shows the pseudo code for this component.

Algorithm 17 Entry Point Detector - process_impl

```
1: procedure PROCESS_IMPL
2:     Get the component data object from the input data source.
3:     if target file name exists then
4:         Get the target file name from the component data object
5:     else
6:         Throw an exception
7:     end if
8:     Get the file type from the meta object found in the component data object.
9:     Get the correct Code_Section_Detector_Algorithm object from the based on the
   file type from the Code_Section_Detector_Algorithm_Factory.
10:    Execute the algorithm using the target file name
11:    Store the returned meta object in the output data source.
12: end procedure
```

Each of the Entry Point Detector algorithms uses its associated file reader to obtain the entry point information. Windows PE, Linux ELF and Mac OS.X MachO all have a field in their header which marks the address in memory of the first instruction to be executed. Java class files have Code Attribute sections that are part of each method. A Java class file without the standard main function is similar to a DLL under Windows or a shared library on Linux. There are many entry points but the system is vastly different. In Linux or Windows a file can have one (EXE/DLL) or many (DLL) entry points. Instead of addresses of where entry points for functions stored in the file Java Class files have a Code Attribute section for each method.

9.4.5 File Header Printer

The File Header Printer obtains a text string representing all the information that can be obtained from the target binary executable. The File Header Printer uses the input file name obtained from the data source and the appropriate file reader produced by the file reader factory to read the necessary fields. This information is displayed to the standard output on the command line. Algorithm 18 shows the pseudo code for this component.

Algorithm 18 File Header Detector - process_impl

```
1: procedure PROCESS_IMPL
2:     Get the component data object from the input data source.
3:     if target file name exists then
4:         Get the target file name from the component data object
5:     else
6:         Throw an exception
7:     end if
8:     Get the correct file reader from the Reader Factory.
9:     Print the file header information retrieved from the file reader to the console
10:    Copy the input meta information to the output data source
11: end procedure
```

9.4.6 File Type Detector

The File Type Detector determines the file type associated with a binary executable. The File Type Detector uses the input file name obtained from the data source and the appropriate file reader produced by the file reader factory to read the necessary fields. This information is appended to the output data source as meta information. Algorithm 19 shows the pseudo code for this component.

Algorithm 19 File Type Detector - process_impl

```
1: procedure PROCESS_IMPL
2:     Get the component data object from the input data source.
3:     if target file name exists then
4:         Get the target file name from the component data object
5:     else
6:         Throw an exception
7:     end if
8:     Get the correct file reader from the Reader Factory.
9:     Store the target file name in the output data source.
10:    Stored the file type information retrieved from the file reader in the output data
       source.
11: end procedure
```

9.4.7 Memory Map Producer

The Memory Map Producer creates a Memory Map object representing the memory image of the the target binary executable as it would look after the operating system

loader completed loading the binary. The Memory Map Producer uses the input file name obtained from the data source and the appropriate file reader produced by the file reader factory to obtain the Memory Map. The Memory Map object is placed inside a data source and passed onto the next Component. Algorithm 20 shows the pseudo code for this component.

Algorithm 20 Memory Map Producer - process_impl

```
1: procedure PROCESS_IMPL
2:     Get the component data object from the input data source.
3:     if target file name exists then
4:         Get the target file name from the component data object
5:     else
6:         Throw an exception
7:     end if
8:     Get the correct file reader from the Reader Factory.
9:     Store the target file name in the output data source.
10:    Stored the Memory Map object retrieved from the file reader in the output data
   source.
11: end procedure
```

9.4.8 Null

The Null Component is a useful Component to condense meta information from various sources. It is primarily used with in Libreverse as a source of information for a component graph and when there is no action required at spots in a formula file. Algorithm 21 shows the pseudo code for this component.

Algorithm 21 Null Component - process_impl

```
1: procedure PROCESS_IMPL
2:     Get the component data object from the input data source.
3:     Copy input data to the output data in the data source.
4: end procedure
```

9.4.9 Tevis Unknown Region Checker

The Tevis Unknown Region Checker implements the security check in Jay-Tevis's dissertation [59] which determines to see if there are any sections in memory not explicitly covered in the Windows PE file header. The original work was expanded to the Linux ELF file format. Java Class format was excluded due to the design of the file format. There is no possibility of a Java Class file having any region in memory not captured somehow in the file format. Algorithm 22 shows the pseudo code for this component.

Algorithm 22 Tevis Unknown Region Checker - process_impl

```
1: procedure PROCESS_IMPL
2:     Get the component data object from the input data source.
3:     if target file name exists then
4:         Get the target file name from the component data object
5:     else
6:         Throw an exception
7:     end if
8:     Get the file type from the meta object found in the component data object.
9:     Get the correct Tevis_Unknown_Region_Checker_Algorithm object from the
   based on the file type from the Tevis_Unknown_Region_Checker_Algorithm_Factory;
10:    Execute the algorithm using the target file name
11:    Store the returned meta object in the output data source.
12: end procedure
```

9.4.10 Tevis Zero Filled Checker

The Tevis Zero Filled Checker implements the security check in Jay-Tevis's dissertation [59] which determines to see if there are any sections in memory that have more than 50 consecutive '0' bytes. The idea is that a region with too many non-zero bytes could be used to store a malicious program. The original work was expanded to the Linux ELF file format. Java Class format was excluded due to the design of the file format. There is no possibility of a Java Class file having any region in memory not captured somehow in the file format. Algorithm 23 shows the pseudo code for this component.

Algorithm 23 Tevis Zero Filled Checker - process_impl

```
1: procedure PROCESS_IMPL
2:     Get the component data object from the input data source.
3:     if target file name exists then
4:         Get the target file name from the component data object
5:     else
6:         Throw an exception
7:     end if
8:     Get the file type from the meta object found in the component data object.
9:     Get the correct Tevis_Zero_Filled_Checker_Algorithm object from the based on
       the file type from the Tevis_Zero_Filled_Checker_Algorithm_Factory;
10:    Execute the algorithm using the target file name
11:    Store the returned meta object in the output data source.
12: end procedure
```

9.4.11 Compiler Classifier

The Compiler Classifier is used to determine the compiler that produced the target binary executable. The Compiler Classifier uses the input file name obtained from the data source and the Compiler Classifier, as described in Chapter 5, to obtain the meta object output. The meta object placed inside a Data Source as meta information. and passed onto the next Component. Algorithm 24 shows the pseudo code for this component.

Algorithm 24 Compiler Classifier - process_impl

```
1: procedure PROCESS_IMPL
2:     Get the component data object from the input data source.
3:     if target file name exists then
4:         Get the target file name from the component data object
5:     else
6:         Throw an exception
7:     end if
8:     Get the directory where the GRNN data is stored
9:     if GRNN data file does not exists then
10:        Throw an exception
11:    end if
12:    Get the file type from the meta object found in the component data object.
13:    Get the correct Classifier_Algorithm object from the based on the file type from
       the Compiler_Classifier_Algorithm_Factory;
14:    Execute the algorithm using the GRNN data file
15:    Store the returned meta object in the output data source.
16: end procedure
```

Chapter 10

Component Graph

The Component Graph is a directed graph of Component objects which represents the reverse engineering work to be done on the target file.

10.1 Graph

The Boost Graph library was used to construct this data structure. Boost has an adjacency list [52], a two-dimensional graph structure, that has an entry for each vertex. Each vertex in the adjacency list contains a list of vertex references for those vertexes that have a link between them. In the case of Libreverse the adjacency list was specialized as a directed graph. Therefore the links between the vertices's goes from the parent vertex to its children only. Each vertex in the directed graph is a pointer to a Component object.

10.2 ID Map

Since the parent of a vertex has a lower unique identifier than its children the graph of Component objects can be sorted in ascending order. The ID Map is a standard template library (STL) map that uses the unique identifier for a Component object as a key and handle to the vertex of where that Component object is in the directed graph. This data structure allows Libreverse to quickly get access to a particular component in the directed graph using only its unique identifier.

10.3 Component Map

The Component Map is a Boost property map that is used to store the Component object at a particular vertex. The [52] states that:

The Boost Property Map concepts define a general purpose interface for mapping key objects to corresponding value objects, thereby hiding the details of how the mapping is implemented from algorithms. The implementation of types fulfilling the property map interface is up to the client of the algorithm to provide. The property map requirements are purposefully vague on the type of the key and value objects to allow for the utmost genericity in the function templates of the generic library.

10.4 Visitor

The Graph Visitor is where the graph processing is controlled. The visitor is a breadth-first search visitor that takes the directed graph, sorts the vertexes in ascending order by their unique ids, calls *run()* on each of the vertexes in turn and passing on the results to the children vertexes. A breadth-first search visitor is the best solution since all the parent vertexes for a target vertex must have completed their work before the child Component can execute. A depth-first search would require some mechanism to revisit a child Component if it had more than one parent. Algorithm 25 shows the pseudo code for the Graph Visitor.

Algorithm 25 Graph Visitor - visit

```
1: procedure PROCESS_IMPL
2:     Get a handle to the graph object from the Component Graph
3:     Topologically sort the vertexes
4:     for each vertex in the Component Graph do
5:         Call run on the Component Object stored at this vertex
6:         Get the number of children for the vertex
7:         Save result from Component Object for each children in the data map.
8:         for each child of the vertex do
9:             Update child Component Object to indicate parent Component Object
   has provided data.
10:        end for
11:    end for
12: end procedure
```

Chapter 11

Data Source

Components in the component graph need to pass information from one to another in such a way that limits the connection between the two. Past solutions hard wired the various parts of the reverse engineering process together. Their design was focus around solving one problem.

There are three typical method for passing information from one part of an application to another. Most applications are designed to pass information with system memory allocated to the application. Second method for passing information can be done through specially formatted files. Finally information can be stored within a database. In order to be flexible to the needs of various users the data sources were configured to allow different formats to be utilized within the Libreverse architecture. At the present time only the memory data source has been implemented. Figure 11.1 shows the class diagram for the data source.

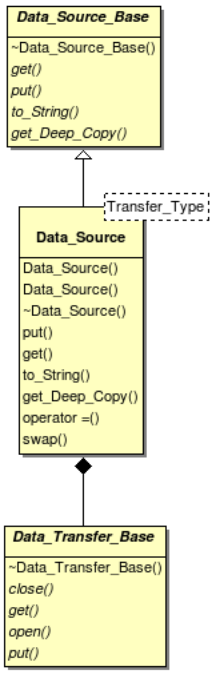


Figure 11.1: Data Source Class Diagram

Each type of data source has a data transfer base object that contains the information to be passed. The data source is explicitly instantiated with the data transfer base class describing the data transfer method. At present only the memory data transfer class is fully implemented in Libreverse.

Each class that inherits from the data transfer base class has a data source config base object containing the information necessary to access the target data. This data source config base object is passed to the data transfer class when it is constructed. Each call to *get()* will return the stored information and *put()* will store the information within the data source. An example is shown in Figure 11.2 for the memory data transfer.

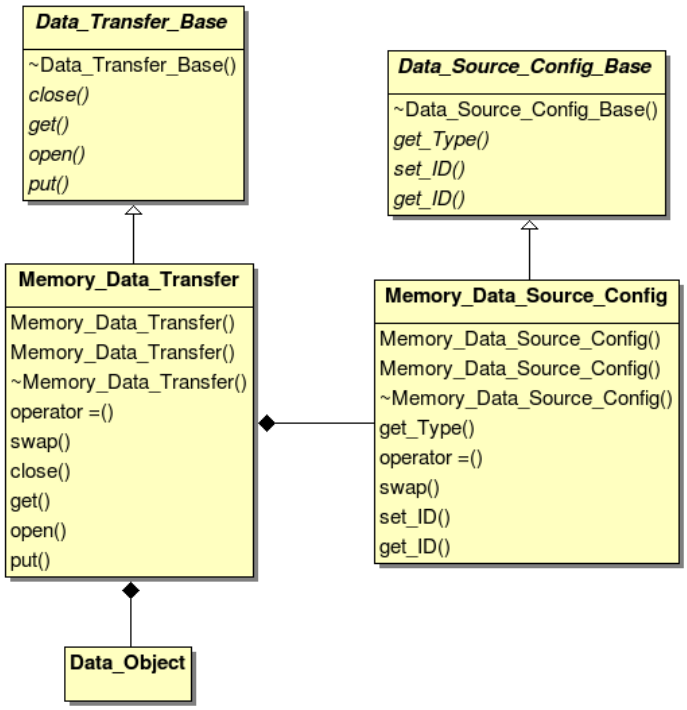


Figure 11.2: Data Transfer Class Diagram

At present the Libreverse architecture supports passing information within system memory allocated to the application. This design decision was used to simplify the implementation of this prototype. The memory data source config object contains the unique identifier for the Component that provides the information. In the operation of the memory data transfer object the *put()* call stores the passed in data object in system memory. The *get()* call returns the store data. The sequence of these operations are shown in Figure 11.3. At present the memory data source config object does not do anything but is in place in case additional information is required to access the memory (e.g. encryption key).

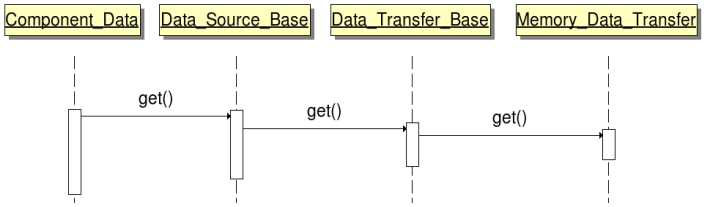


Figure 11.3: Memory Data Transfer Sequence Diagram

The key design principle is the separation of what is being store, data object, how it is being transferred, data transfer base, from how it is configured, data source config base, allows for great flexibility in future designs. More on how the future design of the data source is covered in Section 15.2.6.

Chapter 12

Data Structures

There are three different types of information commonly passed between Components.

12.1 Filename

Some components require the filename of the target file so that they can obtain a handle to the appropriate Reader. These components use the Reader to read values from the header associated with the target file or gain access to the Memory Map for the target file image. The filename data structure is a string that represents the full path to the target file. This is shown in Figure 12.1.

12.2 Memory Map

A memory map object represents a image of a section of memory. It is composed of an index of where the read-write head is in the image, an index of where the read-write head was last located, the base address of the image and vector of bytes. When the memory map is created it is initialized for the desired size and base address. A Component can seek to particular addresses or indexes within the memory map to access data. For example a Memory map allows access within the range of 0 to N where N is the last byte (e.g. memory_map.index_Seek (10)) followed by a read. Or the component can go to a particular address and read information (e.g. memory_map.address_Seek (0x80048f8f)) followed by a read. There is not a list of addresses associated with each index in the memory map. The address associated with each byte is used to calculate the exact index where the read needs to start. For example, if a memory map has a size of 10, a base address of 0x5000 and follows a address_Seek to 0x5005 the index of the

target memory is 0x5005 - 0x5000 = 5. By using a calculation of the index to a byte of memory does not require the memory map to contain a map of addresses with their associated index. This is shown in Figure 12.1.

12.3 Control Flow Graph (CFG) Sequence

A control flow graph sequence is a collection of control flow graphs. A executable program has one entry point where the operating system begins to execute the instructions. The CFG Sequence associated with this program will only contain one CFG. Whereas a shared library will have one or more entry points depending on which functions are publicly available. The CFG Sequence associated with the shared library will have one or more CFG. Each CFG will represent one function in the shared library. Each CFG is similar to the component graph. Each node in the CFG has a unique id and a basic block object associated with it. Each basic block contains a list of instructions associated with that part of the program code before a jump instruction. This is shown in Figure 12.1.

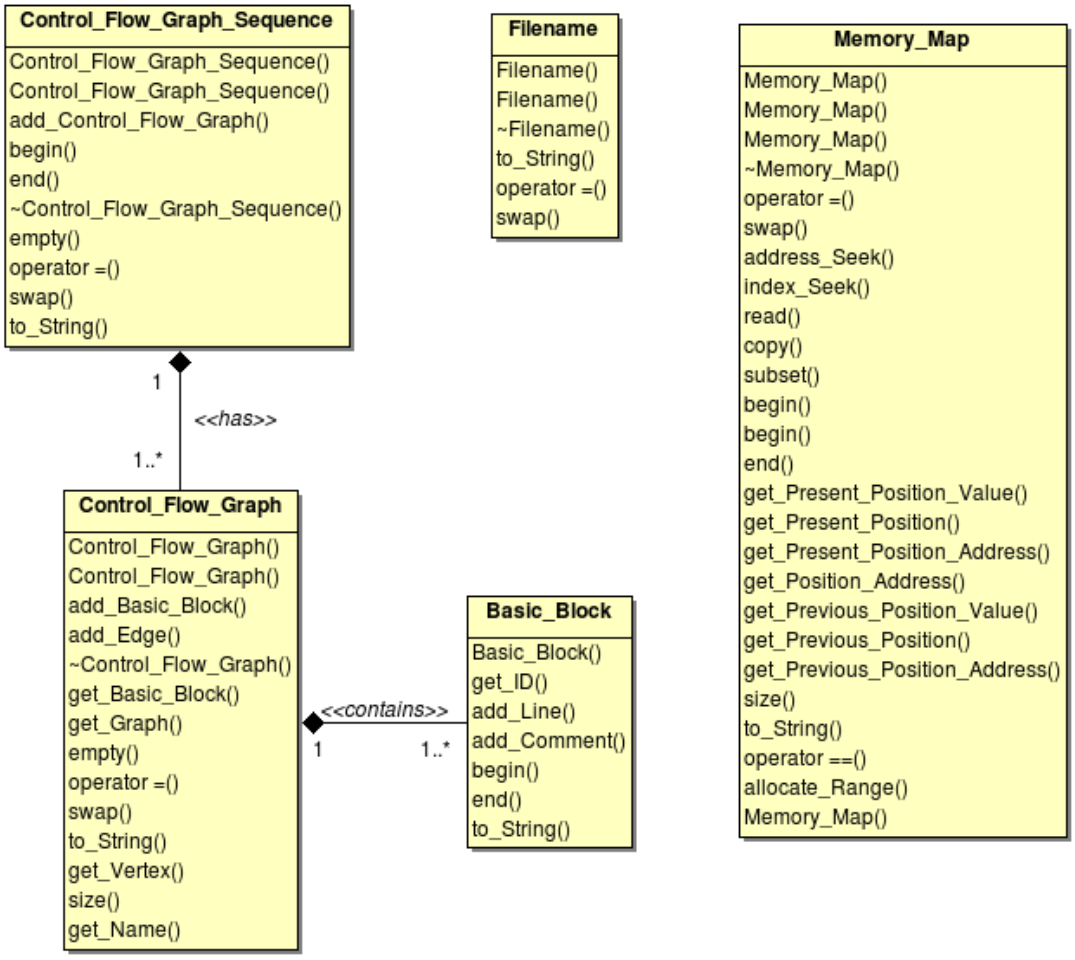


Figure 12.1: Data Container Class Diagram

Chapter 13

Meta Information

13.1 Importance of meta information

The properties that can be obtained from various components describing the target binary are called meta information. Previous approaches to reverse engineering limited their scope to a particular type of a target binary. Meta information significantly reduces and/or eliminates the need to have special application to reverse engineering a target binary.

When constructing a generic reverse engineering architecture it is necessary to design the system in such a way as to eliminate the number of custom components for a particular analysis. Consider the situation where Libreverse would be used to detect which compiler was used for a target binary. If there was not a means to provide extra information about a target binary then there would need to be unique solutions for each type of target binary. Each unique solution would have to have two steps: detection of the target file type and detection of the compiler used. For example, for Windows PE there would be a component to verify that the target binary was indeed a Windows PE file. There would also be a component to perform the compiler detection for compilers that produce Windows PE files. If there were only a few kinds of compilers and target binary file formats then have unique solutions would not be that much of an issue. Unfortunately, since new file formats are coming out as operating systems are created or extended every year this solution does not scale well. Figure 13.1 visual shows how with just three target binary file formats the problems with the unique component approach become apparent.

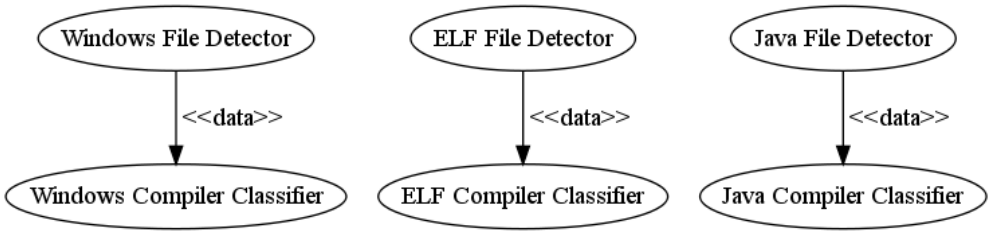


Figure 13.1: Three sets of unique components for handling compiler classification

It is important to note that there are overlapping steps performed with each unique component. Each file detector requires a file reader that enables it to gather information from the file. The file detector produces an answer to indicate that it was successful or not. This data is passed on to the next component for analysis. In addition each compiler classifier also requires a file reader to gather information as the file detector and specific actions to classify the target binary's compiler.

This solution can be simplified down from the three different sets of components into one set of components. The first component is a file type detector that uses the data, target binary, to determine what its file type. It passes this newly discovered information as meta information to the next component. The second component reads the meta information, determines the algorithm it needs to use, executes the algorithm and passes on the results. This can be seen in Figure 13.2

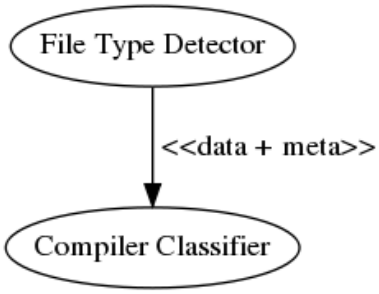


Figure 13.2: Generic components for handling compiler classification

13.2 How meta information is exchanged

Meta information is passed along in a meta object with a Data Source providing Components with hints about the kind of binary file being analyzed. These hints can allow the designer of the Component to alter the behavior for specific actions. It is important to realize that different reverse engineering applications have similar actions that are performed. They only difference between these applications is the type of binary being analyzed. So rather than having N different kinds of reverse engineering tools one solution can be used to dynamically change its behavior at run-time.

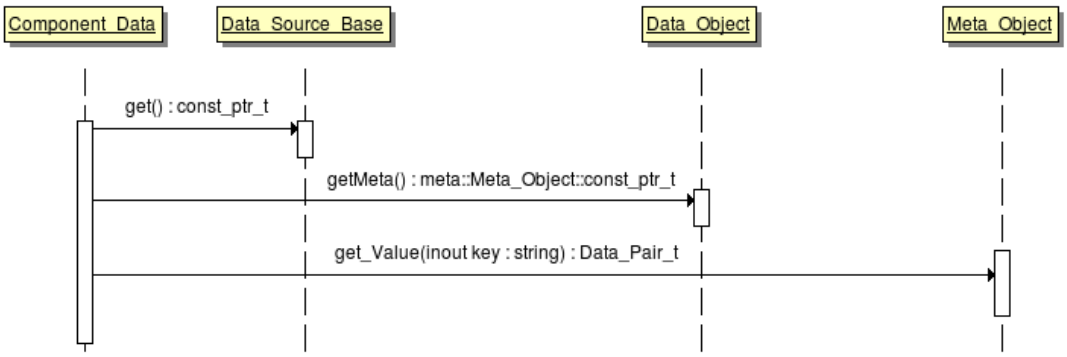


Figure 13.3: Reading meta information from a Data Source

Figure 13.3 shows the steps in obtaining a meta entry. Each Data Source contains one meta object storing all the meta information received thus far in the reverse engineering process. A component will condense the meta information obtained from each received Data Source into one meta object before beginning its work. The first step is to call 'get()' from each of the Data Sources to get a handle on the underlying Data Object. How the Data Object is stored is described in more detail in Chapter 11. Once a component has a handle to the Data Object it can access the meta object, shown in Figure 13.4, for the meta information stored in it. Each entry in a meta object is a key with a matched Meta Item object. The key is used to access the right Meta Item if it is present in the meta object. The Meta Item can represent a variety of different

kinds of meta information as described in Section 13.3. After obtaining the desired meta information the component can use this information to customize its behavior.

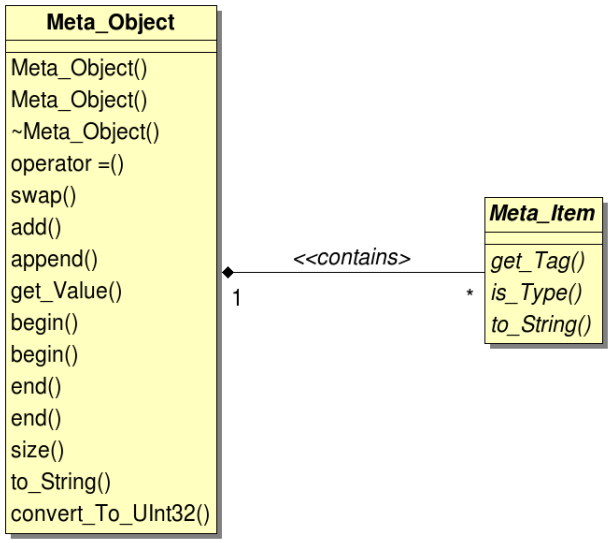


Figure 13.4: Meta Object class diagram

By decomposing complex operations into a set of components that use meta information to select the correct algorithm to use for a specific situation it eliminates the need for a number of custom components for a particular analysis. Libreverse implements this design criteria thereby enabling the architecture to be easily expanded to new kinds of meta information and components. This is one of its great strengths when compared to the specific solutions mentioned in Chapter 2. Each of the specific solutions have common actions requiring specific information necessary to perform their purpose. These common actions can be decomposed into simple steps using meta information to alter their behavior. For example, Cifuentes [14] decompiled MS-DOS x86 programs into source files written in C programming language. Each of the steps performed could be made into simple components using and providing meta information to alter Libreverse's run-time behavior. It would be easy to extend the work to handle Java class files as input and output them as source files written in the C programming language.

13.3 Meta Items

The meta item class and the child classes which inherit from it were modeled after how the java class file format stores attribute information. In the java class file format a generic base class is used to hide the kind of attributes listed in the file header. If the JVM understands the attribute, identified by a unique tag, then the attribute is utilized otherwise it would silently ignore it. The important principle inspired by the java class format was to hide the kind of meta item objects passed. A meta object has no knowledge of the kind of meta items it contains. Each component in Libreverse uses the the appropriate meta item tag to retrieve the meta item from the meta object. This section describes each of the meta item types shown in Figure 13.5.

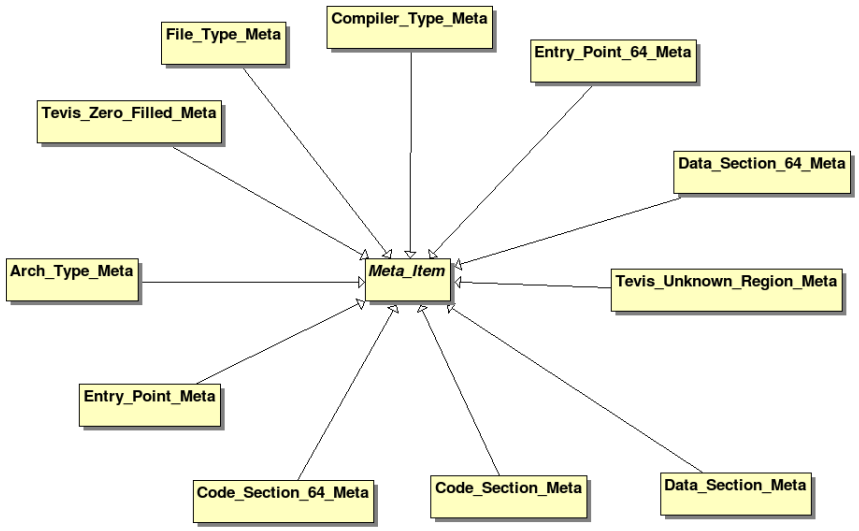


Figure 13.5: Meta Item class diagram

13.3.1 Arch Type Meta

The Arch Type Meta, shown in Figure 13.6, is produced by the Architecture Type Detector, described in Section 9.4.1, to capture the CPU architecture supported in the target file. The 'm_tag' is the unique identifier, 'm_bit_length' captures whether the architecture is 32/64 bit, and the 'm_type' shows the supported architecture.

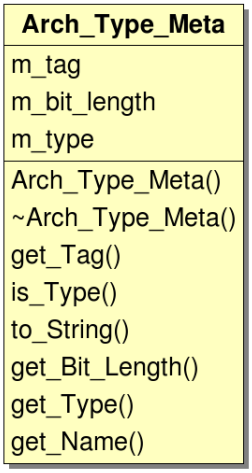


Figure 13.6: Arch Type Meta class diagram

13.3.2 Code Section Meta

The Code Section Meta, shown in Figure 13.7, is produced by the Code Section Detector for 32-bit files, described in Section 9.4.2, to determine the beginning address of the code section in a 32-bit target file. The 'm_tag' is the unique identifier, 'm_offset' is the offset of the code section from the beginning of the memory for the target file, and 'm_size' is the length of the section.

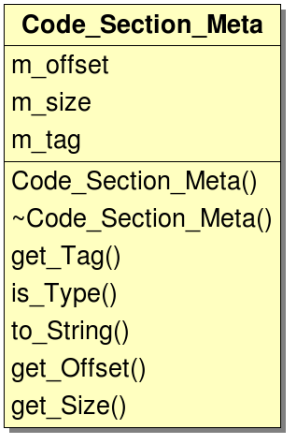


Figure 13.7: Code Section Meta class diagram

13.3.3 Code Section 64 Meta

The Code Section 64 Meta, shown in Figure 13.8, is produced by the Code Section Detector for 64-bit files, described in Section 9.4.2, to determine the beginning address of the code section in a 64-bit target file. The 'm_tag' is the unique identifier, 'm_offset' is the offset of the code section from the beginning of the memory for the target file, and 'm_size' is the length of the section.

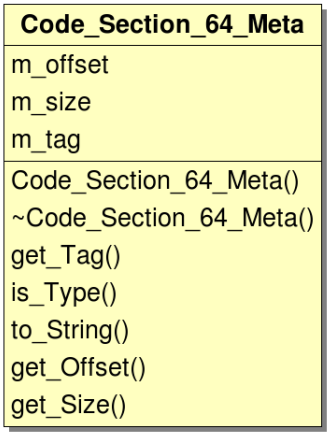


Figure 13.8: Code Section 64 Meta class diagram

13.3.4 Compiler Type Meta

The Compiler Type Meta, shown in Figure 13.9, is produced by the Compiler Type Detector for 32-bit files, described in Section 9.4.11, to determine the compiler used to create the target file. The 'm_tag' is the unique identifier, and 'm_compiler_id' is the unique id describing the detected compiler.

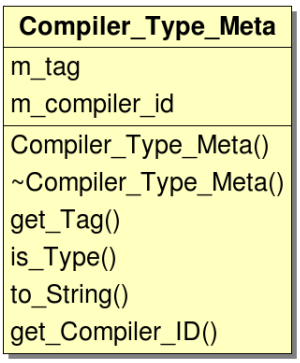


Figure 13.9: Compiler Type Meta class diagram

13.3.5 Data Section Meta

The Data Section Meta, shown in Figure 13.10, is produced by the Data Section Detector for 32-bit files, described in Section 9.4.3, to determine the beginning address of the data section in a 32-bit target file. The 'm_tag' is the unique identifier, 'm_offset' is the offset of the data section from the beginning of the memory for the target file, and 'm_size' is the length of the section.

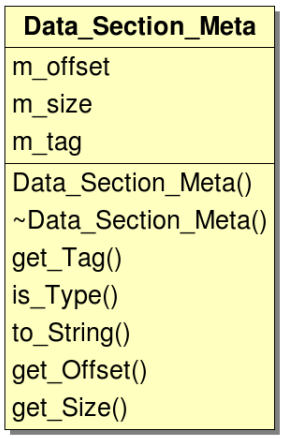


Figure 13.10: Data Section Meta class diagram

13.3.6 Data Section 64 Meta

The Data Section Meta, shown in Figure 13.11, is produced by the Data Section Detector for 64-bit files, described in Section 9.4.3, to determine the beginning address of the data section in a 64-bit target file. The 'm_tag' is the unique identifier, 'm_offset' is the offset of the data section from the beginning of the memory for the target file, and 'm_size' is the length of the section.

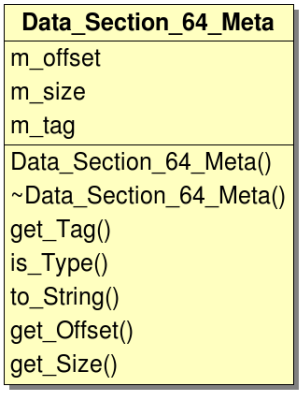


Figure 13.11: Data Section Meta class diagram

13.3.7 Entry Point Meta

The Entry Point Meta, shown in Figure 13.12, is produced by the Entry Point Detector for 32-bit files, described in Section 9.4.4, to determine the list of addresses describing the entry point in a 32-bit target file. The 'm_tag' is the unique identifier and 'm_entry_points' is the list of entry points.

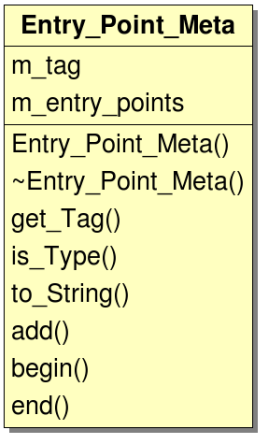


Figure 13.12: Entry Point Meta class diagram

13.3.8 Entry Point 64 Meta

The Entry Point Meta, shown in Figure 13.13, is produced by the Entry Point Detector for 64-bit files, described in Section 9.4.4, to determine the beginning address of the entry point in a 64-bit target file. The 'm_tag' is the unique identifier and 'm_entry_points' is the list of entry points.

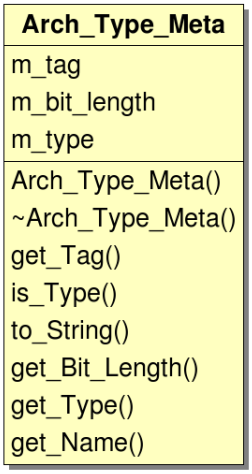


Figure 13.13: Entry Point Meta class diagram

13.3.9 File Type Meta

The File Type Meta, shown in Figure 13.14, is produced by the File Type Detector, described in Section 9.4.6, to determine the file type of the target file. The 'm_tag' is the unique identifier, 'm_bit_length' is the bit length of the target file, and 'm_type' is the description of the file type.

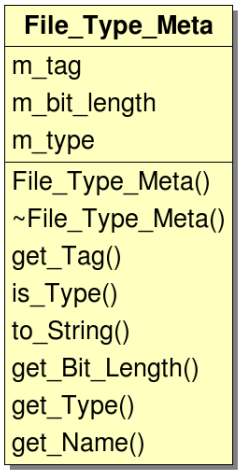


Figure 13.14: File Type Meta class diagram

13.3.10 Tevis Unknown Region Meta

The Tevis Unknown Region Meta, shown in Figure 13.15, is produced by the Tevis Unknown Region Detector, described in Section 9.4.9, to determine the regions in memory of a target file not described in the file headers. The 'm_tag' is the unique identifier and 'm_unknowns' is the list of unknown regions found in the file.

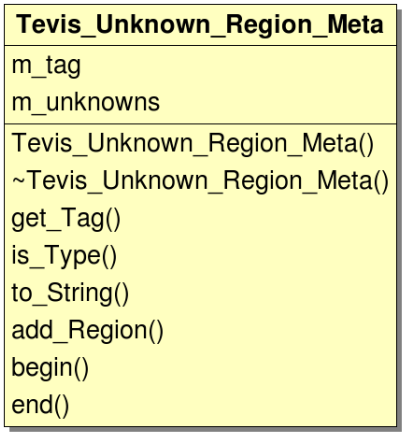


Figure 13.15: Tevis Unknown Region Meta class diagram

13.3.11 Tevis Zero Filled Meta

The Tevis Zero Filled Meta, shown in Figure 13.16, is produced by the Tevis Zero Filled Checker, described in Section 9.4.10, to determine the areas in memory for target that have more that 50 consecutive zeroes. The 'm_tag' is the unique identifier and 'm_zero_filled' is the list of regions found in the file that have more than 50 consecutive zeros.

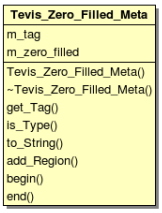


Figure 13.16: Tevis Zero Filled Meta class diagram

Chapter 14

Scenarios

In order to best understand the impact of this research two scenarios are used throughout it to provide two different points of view. The problem of merely presenting the research is that it does not connect the reader with the vision of the research. It fails to provide the extra information that lives inside the mind of the researcher.

14.1 Non-expert and Expert user

The non-expert user of Libreverse is a 15-year veteran of a federal agency. His job in this division is to take suspicious programs off suspect's computers and attempt to classify the programs for other investigators. Due to a rash of new white collar crimes the department is looking for a automated method to assist the non-expert user. The department wants a tool that can provide information on what compiler was used and if any compression utility was applied to the file.

The expert user of the Libreverse is a 20-year computer security researcher for a defense contracting company who is working with the federal agency on an assignment. The expert user has been using the Libreverse for years and now would like to add a new compiler classification. The expert user is known as an expert in the field of compression and compiler classification using artificial intelligence (AI). The expert user will be configuring Libreverse reverse so that the non-expert user can perform their task without requiring extensive education.

14.2 Compressed Executables and Compiler Classification

In this problem the novice user utilizes the application supplied by the expert user to process each suspect file. The novice user is not making any assumptions about

the suspect files found on a suspect's computer. They are relying on Libreverse to process each suspect file. From their point of view they use a program supplied by the expert user to process a set of files. This application interacts with Libreverse to perform the necessary analysis for each file. Once the novice user starts the application their interaction with Libreverse is done. All they has to do is wait for the result. In the arrangement of components there is a component that will classify the compiler use to create the suspect file or determine. The compiler classification component uses the original file name to determine the compiler used. The result of this component is new meta information stating the compiler id used to create the suspect file. This meta information would be utilized by other components to alter their behavior. In the novice user case the output information is used by his application to move the suspect file into a preset directory. In a similar way the compressed executable detector, which will provide meta information on whether any sort of compression was used on the file. This information will be used to place the suspect file into a present directory for further analysis.

During a recent investigation into software piracy the expert user discovered a binary executable with a new type of compression. Further investigation of the suspect computer uncovered the new compression utility. The expert user will take the new compression program and compress a predefined set of test programs. After which they will collect the necessary attribute input data into a file for classification and rerun the optimizer. The resulting attributes used to classify between the various compression methods will be used to alter the compressed executable detector component.

In addition the expert user wants to expand Libreverse's ability to detect a new compiler he has found on a suspect's computer running Windows Vista. Libreverse's compiler classification has no knowledge of this new compiler. At this point the expert user has two choices. First, they can either create a whole new program that can classify this new compiler along with those that Libreverse knows. Or they can take the same steps to classify this new compiler as done with previous compilers that produced Windows PE files. The former method seems to be the way a lot of reverse engineering

projects take. A specific program is created for a specific problem. The downside is this approach is that the program cannot be reused. Since the expert user values their time they know that all they have to do is to compile a set of test programs with the new compiler. The expert user updates the trainer_dump application from Libreverse to read the necessary attributes used for classification from the new set of test programs. This extracted information is placed in the same directory as the other input data collected for compilers producing Windows PE executables. After executing trainer_dump on the new files they update the grnn_optimizer to take advantage of this new data. At this point the expert user can run the grnn_optimizer to discover which attributes are necessary to classify between the old Windows compilers and the newly discovered compiler. the expert user takes this information and updates the compiler classification component.

14.3 Behind the scene

The novice user's analysis program communicates with Libreverse via the *execute* function in the API. The analysis program will pass along the string name for the path to the target file along with the suspected input type (BINARY) and output type (CPLUSPLUS). It will not utilize any of the tracing parameters since those are mainly used for debugging Libreverse.

Similarly the expert user uses the same analysis program as the novice user. The only difference is the expert user requires different output (e.g. UML) and utilize the tracing parameters so he can follow Libreverse as it attempts to use his new compiler classification additions.

For each user the configurator takes the given input and output types to look up the appropriate formula list in the master formula map. The master formula map is created from the contents of the master formula file stored on the system. The master formula file used for this case study is shown in Listing 14.1.

Listing 14.1: Case study master formula file

```
<?xml version="1.0"?>
<formula_list>
  <formula>
    <input>binary</input>
    <output>uml</output>
    <input_formula>binary_RTL_2.xml</input_formula>
    <analysis_formula>decompiler_analysis.xml</analysis_formula>
    <output_formula>uml_output.xml</output_formula>
  </formula>
  <formula>
    <input>binary</input>
    <output>C++</output>
    <input_formula>binary_RTL.xml</input_formula>
    <analysis_formula>decompiler_analysis.xml</analysis_formula>
    <output_formula>cplusplus_output.xml</output_formula>
  </formula>
</formula_list>
```

The formulas described in this section are for the purpose of showing the flexibility of the configuration system. Unless otherwise noted in Chapter 9 the components used in the formula are future features of Libreverse not presently implemented.

Both the expert and non-expert user utilize the input formula that converts the binary input into a control flow graph (CFG) in RTL format. The expert user will store an altered input formula file, binary_RTL_2.xml, under a new name to allow testing a compressed executable unpacker component. The input formula, binary_RTL.xml, used by the non-expert user is shown in XML format in Listing 14.2 and graphically in Figure 14.1. The input formula used by the expert user is shown in Listing 14.3 and graphically in Figure 14.2.

Listing 14.2: Case study input formula file for non-expert user (binary_RTL.xml)

```
<?xml version="1.0"?>
<formula_map>
  <node>
    <name id="1">file_type_detector</name>
  </node>
  <node>
    <predecessors><node_ref idref="1"/></predecessors>
    <name id="2">memory_map_producer</name>
  </node>
  <node>
    <predecessors><node_ref idref="1"/></predecessors>
    <name id="3">compiler_classifier</name>
  </node>
  <node>
    <predecessors><node_ref idref="1"/></predecessors>
    <name id="4">arch_detector</name>
  </node>
  <node>
    <predecessors>
      <node_ref idref="2"/>
      <node_ref idref="3"/>
      <node_ref idref="4"/>
    </predecessors>
    <name id="5">control_flow_producer</name>
  </node>
  <node>
    <predecessors><node_ref idref="5"/></predecessors>
    <name id="6">rtl_converter</name>
  </node>
</formula_map>
```

Listing 14.3: Case study input formula file for expert user (binary_RTL_2.xml)

```
<?xml version="1.0"?>
<formula_map>
  <node>
    <name id="1">file_type_detector</name>
  </node>
  <node>
    <predecessors><node_ref idref="1"/></predecessors>
    <name id="2">memory_map_producer</name>
  </node>
  <node>
    <predecessors><node_ref idref="1"/></predecessors>
    <name id="3">compiler_classifier</name>
  </node>
  <node>
    <predecessors><node_ref idref="1"/></predecessors>
    <name id="4">arch_detector</name>
  </node>
  <node>
    <predecessors><node_ref idref="2"/></predecessors>
    <name id="5">compressed_executable_unpacker</name>
  </node>
  <node>
    <predecessors>
      <node_ref idref="3"/>
      <node_ref idref="4"/>
      <node_ref idref="5"/>
    </predecessors>
    <name id="6">control_flow_producer</name>
  </node>
  <node>
    <predecessors><node_ref idref="6"/></predecessors>
    <name id="7">rtl_converter</name>
  </node>
</formula_map>
```

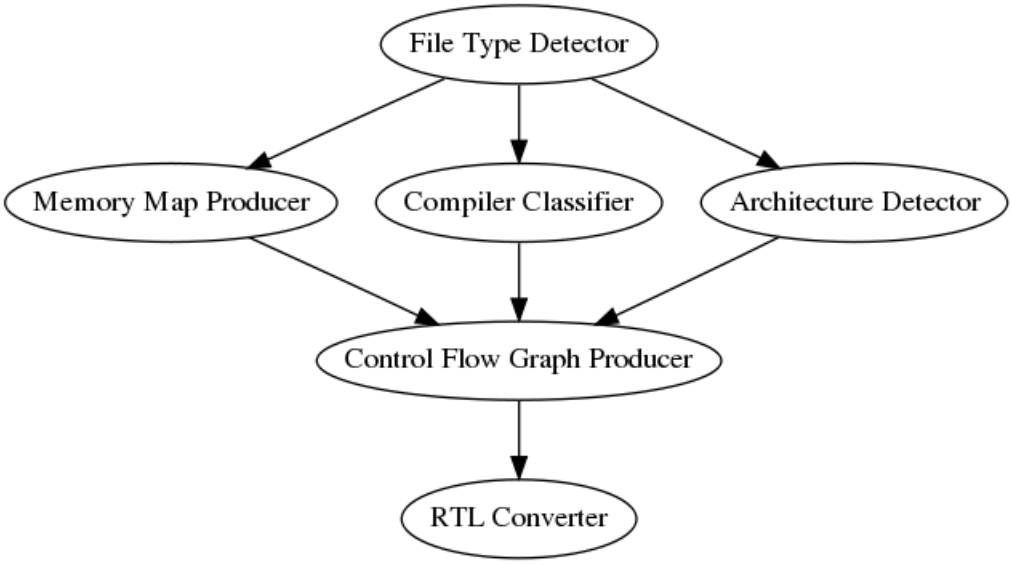


Figure 14.1: Graphical view of binary_RTL.xml

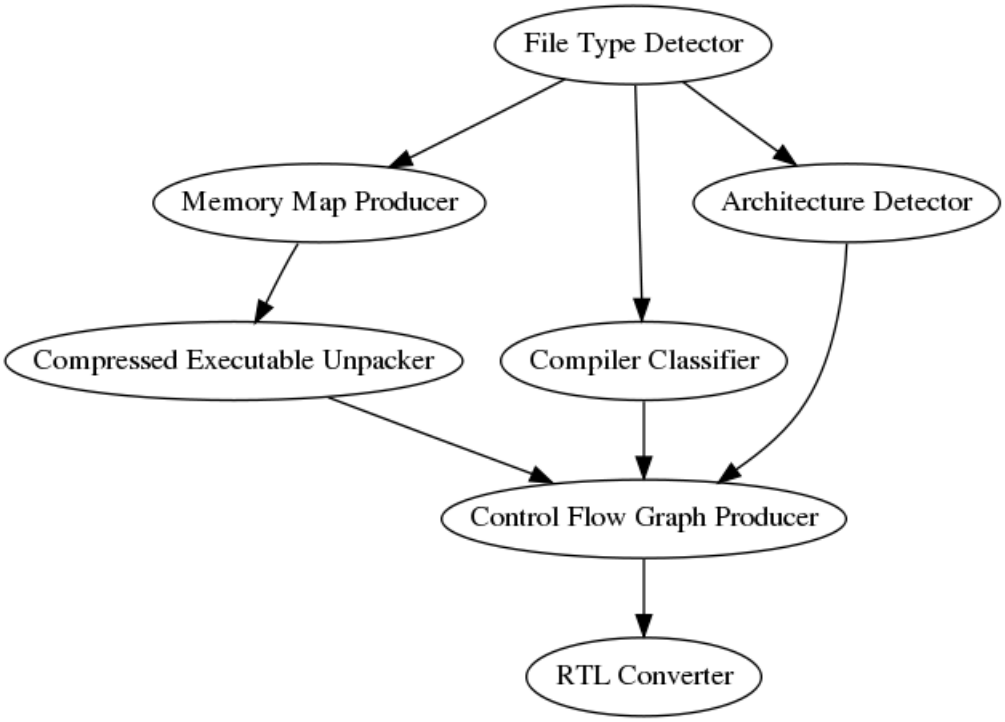


Figure 14.2: Graphical view of binary_RTL2.xml

The configuration system obtains the correct formula list from the master map using the input keys for the type of input and desired output (e.g. BINARY,CPLUSPLUS). It parses the binary_RTL.xml file for the non-expert user to return the component graph of components listes in the XML file. Whereas the configurator will grab the formula list from the master map using the keys BINARY,UML and parse the binary_RTL_2.xml file, shown in Figure 14.2, to return the component graph for the expert user. Each will utilize the same formula file seen in Listing 14.4 and Figure 14.3 for the analysis phase of the reverse engineering process.

The input to the analysis formula map will be the results of the input formula map contained in the last component. The first component to receive the data is the dead register elimination components. Its results are passed onto the dead code elimination and so on. The end result is a CFG containing all the information in a higher level format more resembling source code.

The output formula for non-expert user is described in Listing 14.5 and shown in Figure 14.4. The output formula for the expert user is described in Listing 14.6 and shown in Figure 14.5. The output from the analysis formula map will be the input to the output formula map. The purpose of this stage is to convert the contents of the CFG into a meaningful format requested by the user. In the case of the non-expert user this output will be source files written in the C++ programming language. For the expert user the output will be files describing the CFG contents using UML.

Listing 14.4: Case study analysis formula file for the expert user (decompiling_analysis.xml)

```
<?xml version="1.0"?>
<formula_map>
  <node>
    <predecessors/>
    <name id="1">dead_register_elimination</name>
  </node>
  <node>
    <predecessors><node_ref idref="1"/></predecessors>
    <name id="2">dead_code_elimination</name>
  </node>
  <node>
    <predecessors><node_ref idref="2"/></predecessors>
    <name id="3">condition_code_propagation</name>
  </node>
  <node>
    <predecessors><node_ref idref="3"/></predecessors>
    <name id="4">register_argument_identification</name>
  </node>
  <node>
    <predecessors><node_ref idref="4"/></predecessors>
    <name id="5">function_return_register</name>
  </node>
  <node>
    <predecessors><node_ref idref="5"/></predecessors>
    <name id="6">register_copy_propagation</name>
  </node>
  <node>
    <predecessors><node_ref idref="6"/></predecessors>
    <name id="7">data_type_propagation</name>
  </node>
</formula_map>
```

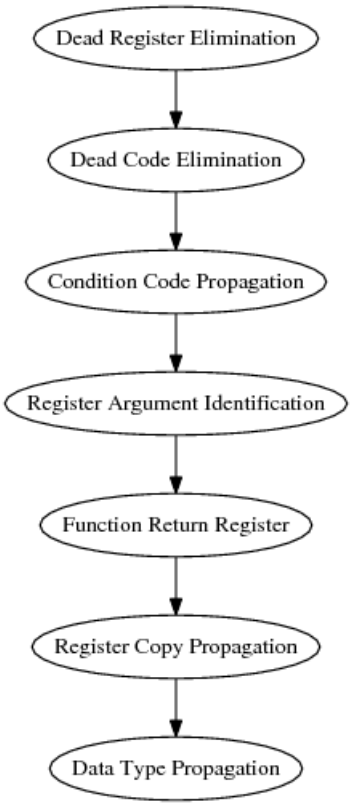


Figure 14.3: Graphical view of decompiling_analysis.xml

Listing 14.5: Case study output formula file for Non-Expert User (cplusplus_output.xml)

```
<?xml version="1.0"?>
<formula_map>
  <node>
    <predecessors/>
    <name id="1">cpp_writer</name>
  </node>
```

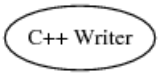


Figure 14.4: Graphical view of cpp_writer.xml

Listing 14.6: Case study output formula file for expert user (uml_output.xml)

```
<?xml version="1.0"?>
<formula_map>
  <node>
    <predecessors/>
    <name id="1">uml_writer</name>
  </node>
```

Figure 14.5: Graphical view of uml_writer.xml

An overview of how all the formulas fit together are shown in Figure 14.6 and 14.7.

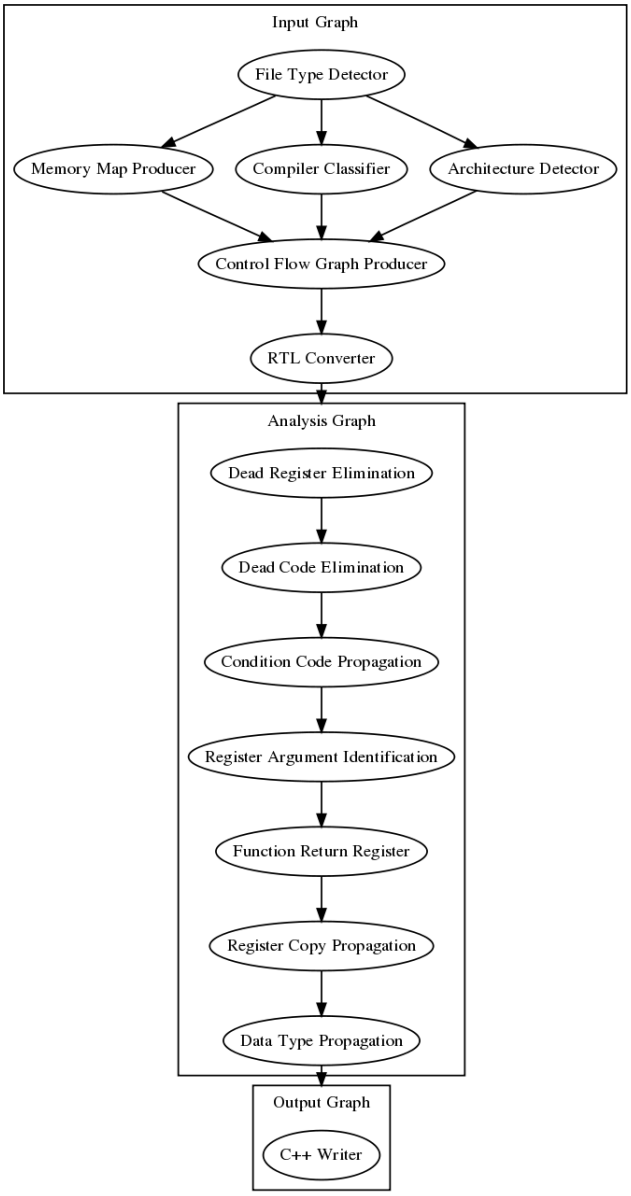


Figure 14.6: Graphical view of Non-Expert User Component Graphs

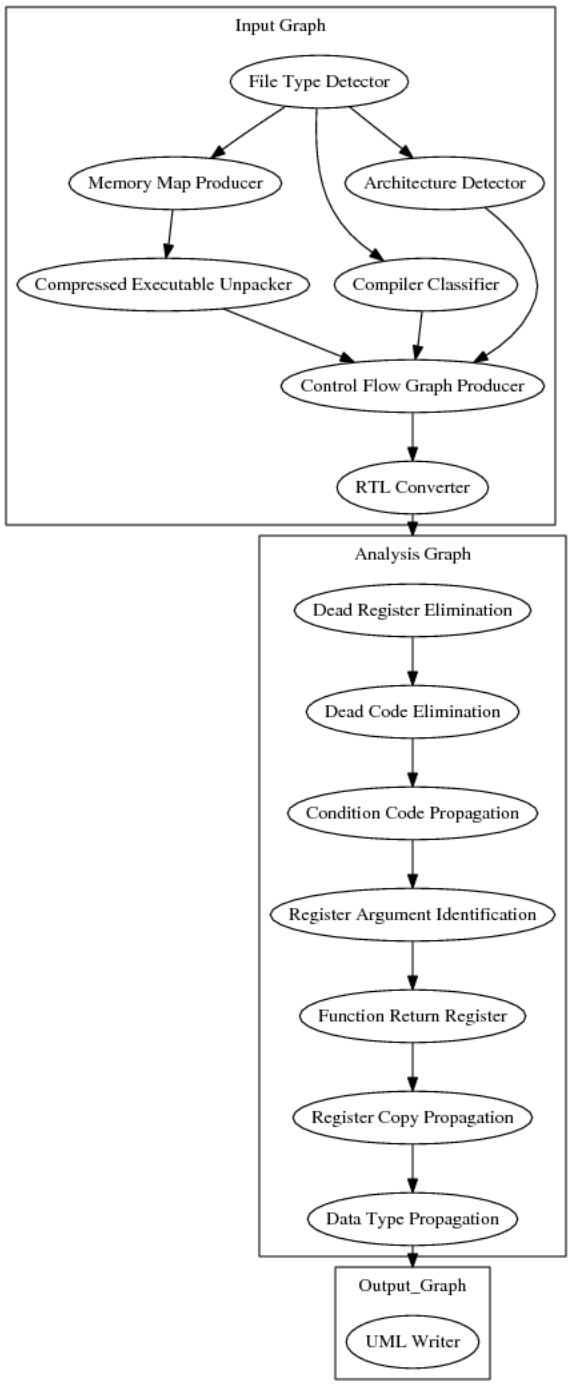


Figure 14.7: Graphical view of Expert User Component Graphs

Chapter 15

Conclusion and Future Work

15.1 Conclusion

In this research we have shown a generic reverse engineering system that is flexible, scalable and able to be alter its behavior based on information detected during the reverse engineering process. In addition this research has shown that it is possible to detect the compiler used to create a binary computer application, regardless of the programming language used, through information gathered from the binary file format by classifying against profiles from a list of available compilers. This work showed that it is possible to classify the compiler used to create an application with 99.88% accuracy for Linux ELF, 99.88% accuracy for Windows PE and 98.53% accuracy for Java Class.

15.2 Future Work

Although we have accomplished a lot during this research the following features are possible for future expansion of Libreverse.

15.2.1 Compiler Detection

The compiler detection needs to be expanded to investigate the idea that it is possible to identify a compiler used from looking at a stream of instructions. This method of compiler detection would be very helpful for the situation where the executable was compressed and/or encrypted after it was compiled. Information in the header might make it difficult or impossible to detect the compiler if key information is altered too much.

15.2.2 Components

In order to show an generic architecture, like Libreverse, is equivalent to a specific solution it must perform all the actions of the specific solution. New components will need to be written to perform the actions done in the work by Cifuentes [14], Tevis [59], Mycroft [44] and others.

Cifuentes used Register Transfer List(RTL) as the intermediate form to represent the code being reverse engineered. Future work will have to first reproduce the original work using RTL but also investigate if the approach of each component, described in Table 15.1 works for other intermediate forms. Table 15.1 describes further components that will be researched and incorporated into the Libreverse reverse engineering architecture.

Tevis conducted his research using the Windows PE format. Linux ELF and Mac OS X Mach-O file format are loaded into memory by the operating system like Windows PE file format. The same methods performed in Tevis [59] should be investigated to see if the same potential security vulnerabilities exist in these other formats. Table 15.2 describes further components that will be researched and incorporated into the Libreverse reverse engineering architecture.

Mycroft [44] investigated a static analysis method of reconstructing the type of registers in target code represented in SSA form. Cifuentes [15] reports that "In von Neumann machines, code and data are represented in the same way, hence making it impossible to distinguish code from data *statically*". Therefore it will be necessary to investigate the dynamic means to reconstruct the type of registers in a program. Table 15.3 describes further components that will be researched and incorporated into the Libreverse reverse engineering architecture.

15.2.3 Control Flow Graph Generation

Reading a stream of assembly code output from a disassembler or a processed assembly output from an application like IDA Pro is still difficult to read. Libreverse needs

Components	Description
Dead-Register Eliminate Dead Registers	Identifies and removes dead registers in the basic blocks of a control flow graph.
Dead-Condition Code Elimination	Identifies and removed condition code from a program if it is never used.
Condition Code Propagation	Each machine instructions set flags in the CPU when they are executed. Therefore where the flags are used and defined must be identified for each machine instruction and used as part of the reverse engineering process.
Register arguments	Identify registers that contain arguments passed for each subroutine.
Function Return Registers	Identify register used to return values from a subroutine.
Register Copy Propagation	Identify temporary registers that are used to hold the contents of other registers. Replace the temporary register name with the original register name.
Actual Function Parameters	Parameters to a function call are either placed into registers or put onto the stack. This component will identify the parameters and place the parameters in the correct order for the function call.
Data Type Propagation Across Procedure Calls	When the type of a register is identified it is necessary to propagate that type to every place the register is used.
Register Variable Elimination	Replace all remaining registers output from the Register Copy Propagation component with variable names.

Table 15.1: Future optimization components from Cifuentes

Components	Description
Detecting Anomalies in the target file	Locate anomalies in the file (e.g. size of table stated in a header from the actual size in the file). These may indicate target file was tampered.
Detect Writable and Executable sections	Locate sections in memory that are both writable and executable.
Detecting Vulnerable Library Functions	Identify functions that are known to be vulnerable to attack.

Table 15.2: Future optimization components from Tevis

Components	Description
Type Reconstruction	Dynamically reconstruct types for registers contained in the instructions basic blocks of a control flow graph.

Table 15.3: Future optimization components from Mycroft

to be extended to contain algorithms that can take the input instructions and produce a control flow graph that can aid in the understanding of a target file. These algorithms need to be able to identify all branch types supported by the input instructions and produce a the necessary control flow graph for future analysis. In addition rather than using a two-step process of converting input instruction to assembly to intermediate form the conversion process should be able to convert from input instruction directly into intermediate form. They type of intermediate form should be an option that the researcher can select as a part of the formula file.

15.2.4 Intermediate Forms

Various researchers use different intermediate forms to perform their analysis (e.g. RTL or SSA). Libreverse needs to be able to represent instructions in

- Assembly language
- Register Transfer Language (RTL)
- Single State Assignment (SSA)

in order to give the researcher a wide choice of methods to represent the target instructions for analysis. By supporting a number of different intermediate forms further research in the components described in Section 15.2.2 will be required to to see if the component's effectiveness is dependent upon the format of its data (e.g. RTL vs SSA).

15.2.5 Configurator

The Configurator part of the infrastructure of Libreverse needs to be expanded to allow for a method to select the data source to use along with new kinds of data sources to transfer information between components.

The Configuration_Data class provides the data structure for storing the configuration information. It allows the setting and retrieval of the transfer configuration type (e.g. Memory), directory where the formula files are store and DLLs. This data structure allows for additional information to be added while keeping the Configurator class free of the accessor methods.

Configurator will be changed to get the type of data source it should use from XML file as shown in Figure 15.1. The limitation is only one transfer type is enforced by the schema allowing only one transfer_type element in the XML file. It can either be a file prefix used in the file name for the output of each Component, database to where data is store or in memory.

Listing 15.1: XML schema for configuration data

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Configuration">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="transfer_type" type="TransferType"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="TransferType">
    <xs:choice>
      <xs:element name="file_prefix" type="xs:string"/>
      <xs:element name="database" type="DatabaseType"/>
      <xs:element name="memory" type="xs:string" nillable="true"/>
    </xs:choice>
  </xs:complexType>

  <xs:complexType name="DatabaseType">
    <xs:sequence>
      <xs:element name="host" type="xs:string"/>
      <xs:element name="user" type="xs:string"/>
      <xs:element name="password" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

15.2.6 Data Source

Two new data sources will be created that allow for the storing of data in a XML file or through a database.

The File data source will store the contents in an XML format using a unique file name. The administrator of Libreverse will use the Configurator's XML file to defined a file prefix to be used. Each file name will be composed of the prefix plus the unique id of the component. A new XML parser will have to be created to read in the file when the requested.

The Database data source will store the contents into a database that was defined by the administrator of Libreverse in the Configurator's XML. A database schema will have to be developed for storing Libreverse's data containers in the database. In addition a

object responsible for creating the data source from the database information or writing to it will also have to be constructed.

15.2.7 Compressed Executables

There are number of opportunities to improve upon Compressed Executable classification to make it more palatable for general use in the field of Computer Forensics:

- The classifier should be able to distinguish between a larger number of compression methodologies.
- These strategies could easily be applied to other operating systems (besides Windows) and file formats, provided that similar representative information exists for those formats as well.
- A separate classifier could possibly be created to further distinguish between types of encryption based on system properties, as demonstrated possible for compression type in Experiment II.
- The methodologies demonstrated here could be implemented in a file analysis framework, for the purpose of computer defense.
- Other classifiers (Radial Basis Function Networks, Feed-Forward Neural Networks, Naïve Bayesian Classifiers) could be implemented and compared with the GRNN.
- Evolutionary Computation could be utilized to create a "White Hat Attacker" which attempted to evolve malicious files which had the signature appearance of non-self-modifying files. The GRNN could then attempt to learn from these "tricky" instances and adapt appropriately by modifying its training data.

15.2.8 Compiler Classification

- Expand the work to distinguish between versions of compilers
- Expand the work to additional different file types (e.g. Mac OS.X MachO and .NET Files)

- Explore alternative machine learning methods, particular those with less overhead.
- Explore identification of compilers using a stream of instructions.

15.2.9 XML

At present the XML library used for parsing the XML files used by libreverse does not validate them against the supplied schema. It is important that the XML files be correct before using them in a production environment. Therefore a method for validating the XML files before they are used needs to be discovered.

15.3 Final Thoughts

So long as there are people attempting to circumvent security controls through malicious software there will be a need to have reverse engineering tools available. These tools will need to be easily adapted to handle the volume of malicious software. They will need to be easily used by the security professional without requiring extensive education into the techniques of reverse engineering. Libreverse provides this necessary flexibility and usability to the non-expert user for processing malicious software as well as a research platform for the expert user. The future of reverse engineering lies in the hands of the dedicated security professionals and researchers protecting our information infrastructures by understanding the software that attacks them and identifying vulnerabilities in the infrastructure software. Reverse engineering will be one line of defense in this constantly shifting battlefield.

Bibliography

[1] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner. State of the Practice of Intrusion Detection Technologies. Technical report. http://www.sei.cmu.edu/publications/documents/99.reports/99tr028/99tr028abstract.html. Last accessed on 9/23/2008., 2000. 4

[2] R. Bellman. *Adaptive Control Processes*. Princeton University Press, Princeton, NJ, 1961. 44

[3] J. Bergeron, M. Debbabi, J. Desharnais, M. Erhioui, Y. Lavoie, and N. Tawbi. Static detection of malicious code in executable programs. In *Proceedings of the International Symposium on Requirements Engineering for Information Security*, 2001. 32

[4] J. Bergeron, M. Debbabi, M.M. Erhioui, and B. Ktari. Static Analysis of Binary Code to Isolate Malicious Behaviors. In *IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 184–189, 1999. 26, 41

[5] P. Biron and A. Malhorta. Xml schema part 2: Datatypes second edition. Technical Report REC-xmlschema-2-20041028, World Wide Web Consortium (W3C), 2004. 92, 94, 96

[6] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. Extensible Markup Language (XML) 1.1 - W3C recommendation 16-august-2006. Technical Report REC-xml11-20060816, World Wide Web Consortium (W3C), 2004. 92, 94, 96

[7] W. Britt, S. Gopalaswamy, J. Hamilton, G. Dozier, and K. Chang. Computer Defense Using Artificial Intelligence. In *SpringSim 07: Symposium on Simulation Software Security*, pages 378–386, 2007. 7

[8] A. Buckley. JSR 202: Java Class File Specification Update. http://jcp.org/en/jsr/detail?id=202. Last accessed on 9/23/2008., 2006. 72, 96

[9] G. Canfora, A. Cimitile, U. De Carlini, and A. De Lucia. An extensible system for source code analysis. *IEEE Transactions on Software Engineering*, 24:721–740, 1998. 8

[10] G. Caprino. Rec: Reverse engineering compiler. http://www.backerstreet.com/rec/rec.htm. Last accessed on 9/23/2008., 2007. 39

[11] E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990. 10

[12] M. Christodorescu and S. Jha. Testing malware detectors. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 34–44, 2004. 5

[13] Z. Chuan, L. Xianliang, H. Mengshu, and Z. Xu. A lvq-based neural network anti-spam email approach. *SIGOPS Operting System Review*, 39(1):34–39, 2005. 7

[14] C. Cifuentes. An environment for the reverse engineering of executable programs. In *APSEC*, pages 410–419, 1995. 8, 29, 33, 41, 42, 113, 120, 147, 170

[15] C. Cifuentes, T. Waddington, and M. Van Emmerik. Computer security analysis through decompilation and high-level debugging. In *In Porceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 375–380. IEEE Press, 2001. 170

[16] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. In *In Logic of Programs. Proceedings of Workshop volume 131 of Lecture Notes in Computer Science*, page 5271. Springer, 1981. 36

[17] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM. 6

[18] L. Davis. *Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991. 54

[19] Un4seen Developments. Petite win32 executable compressor, 2005. Published online at http://www.un4seen.com/petite. Last accessed on 6/20/2008. 65

[20] S. K. Dogra. Normalization, 2008. Published online at http://www.qsarworld.com/qsar-statistics-normalization.php. Last accessed on 8/14/2008. 104

[21] M. Van Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 27–36. IEEE Computer Society, 2004. 31

[22] A. Engelbrecht. *Computational Intelligence*. John Wiley & Sons, Ltd, West Sussex, England, 2002. 2, 5, 69

[23] D. Eriksson. Designing an object-oriented decompiler - decompilation support for interactive disassembler pro. Master's thesis, Blekinge Institute of Technology, June 2002. 34

[24] M. Ernst. Static and dynamic analysis: Synergy and duality. In *Workshop on Dynamic Analysis 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, 2003. 28

[25] D. Fallside and P. Walmsley. Xml schema part 0: Primer second edition. Technical Report REC-xmlschema-0-20041028, World Wide Web Consortium (W3C), 2004. 92, 94, 96

[26] S. Forrest and M. Mitchell. Relative building-block fitness and the building-block hypothesis. In L. Darrell Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 109–126. Morgan Kaufmann, San Mateo, CA, 1993. 52

[27] D. Golberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989. 52

[28] S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 1999. 7, 49

[29] J. Hoenicke. Java optimize and decompile environment (jode). Available at http://jode.sourceforge.net. Last accessed on 9/23/2008., 2006. 40

[30] S. Jarzabek and T. Keam. Design of a generic reverse engineering assistant tool. In *Proceedings of 2nd Working Conference on Reverse Engineering, 1995.*, pages 61–70. IEEE, July 1995. 8

[31] W. Jing-xin, W. Zhi-ying, and D. Kui. A network intrusion detection system based on the artificial neural networks. In *InfoSecu '04: Proceedings of the 3rd international conference on Information security*, pages 166–170, New York, NY, USA, 2004. ACM Press. 7

[32] K. Kapil and V. Kumar. Analysis of binary programs. http://www.cse.iitk.ac.in/report-repository/2005/Y1189_Y1392_BTP_Report.ps. Last accessed on 9/23/2008., April 2005. 33

[33] J. Kinder, S. Katzenbeisser, C. Schallhart, and H. Veith. Detecting malicious code by model checking. In *Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA'05)*, volume 3548 of *Lecture Notes in Computer Science*, pages 174–187. Springer, July 2005. 35

[34] P. Kouznetsovi. JAD - the fast JAva Decompiler. http://www.kpdus.com/jad.html. Last accessed on 9/23/2008., 2001. 38

[35] G. Krol. Boomerang. http://boomerang.sourceforge.net/index.php. Last accessed on 9/23/2008., 2006. 31, 113, 120

[36] S. Kumar. Disc: Decompiler for turboc. Available at http://www.debugmode.com/dcompile/disc.htm. Last accessed on 9/23/2008., 2003. 40

[37] lczelion. Tutorial 6: Import table, 2008. Published online at http://win32assembly.online.fr/pe-tut6.html. Last accessed on 9/24/2008. 17

[38] Microsoft. Prb: Types of thunking available on win32 platforms, 2005. Published online at http://support.microsoft.com/kb/125710. Last accessed on 9/24/2008. 17

[39] Microsoft. Visual Studio, Microsoft Portable Executable and Common Object File Format (Rev. 8.0). http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx. Last accessed on 9/23/2008., 2006. 5, 11, 12, 13, 14, 17, 18

[40] Microsoft. Windows authenticode portable executable signature format, 2008. Published online at http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/Authenticode_PE.docx. Last accessed on 9/24/2008. 16

[41] J. Miecznikowski and L. Hendren. Decompiling java using staged encapsulation. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 368, Washington, DC, USA, 2001. IEEE Computer Society. 37, 120

[42] A. Mori, T. Izumida, T. Sawada, and T. Inoue. A tool for analyzing and detecting malicious mobile code. In *ICSE*, pages 831–834, 2006. 43, 44

[43] S. Mukkamala, A. Sung, and A. Abraham. Cyber security challenges: Designing efficient intrusion detection systems and antivirus tools. http://citeseer.ist.psu.edu/735221.html. Last accessed on 9/23/2008., 2005. 27

[44] A. Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *8th European Symposium on Programming, ESOP'99*, pages 208–224, 1999. 170

[45] A. Neshkov. DJ Java Decompiler. http://members.fortunecity.com/neshkov/dj.html. Last accessed on 9/23/2008., 2007. 39

[46] M. F. Oberhumer and L. Molnár. The ultimate packer for executables (upx), 2005. Published online at http://upx.sourceforge.net. Last accessed on 6/20/2008. 65

[47] M. Pietrek. Programming for 64-bit windows, 2000. Published online at http://msdn.microsoft.com/en-us/magazine/bb985017.aspx. Last accessed on 9/24/2008. 16

[48] S. Russell and P. Norvig. *Artifical Intelligence: A Modern Approach*. Prentice Hall, 2 edition, 2003. 52

[49] H. Schorr. Computer-Aided Digital System Design and Analysis Using a Register Transfer Language. *IEEE Transactions on Electronic Computers*, pages 730–737, 1964. 6

[50] M. Schultz, E. Eskin, E. Zadok, and S. Stolfo. Data Mining Methods for Detection of New Malicious Executables. In *IEEE Symposium on Security and Privacy*, pages 38–49, 2001. 44

[51] A. Shulga. Andromeda Decompiler, 2005. 32

[52] J. Siek, L. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual (C++ In-Depth Series)*. Addison-Wesley Professional, 2001. pp. 213,277-278. 135

[53] D. Specht. A General Regression Neural Network. *IEEE Transactions on Neural Networks*, 2:568–576, 1991. 5, 47, 69

[54] R. Stallman. personal communication, 2008. January 24, 2008 email. 45

[55] A. Sung and S. Mukkamala. Identifying important features for intrusion detection using support vector machines and neural networks. In *Symposium on Applications and the Internet (SAINT)*, pages 209–217, 2003. 29, 45

[56] Symantec. Symantec Security Response - Definitions Added [Online]. http://www.symantec.com/business/security_response/index.jsp. Last accessed on 9/23/2008., 2006. 5

[57] K. Tan. The application of neural networks to UNIX computer security. In *Proc. Int. Conf. Neural Networks, ICNN*, pages 476–481. IEEE Computer Society, 1995. 7

[58] G. Tesauro, J. Kephart, and G. Sorkin. Neural Networks for Computer Virus Recognition. In *IEEE Expert*, volume 11, pages 5–6, 1996. 7, 44

[59] J. Tevis. *Automatic Detection of Software Vulnerabilities in Executables*. PhD thesis, Auburn University, 2005. 42, 132, 170

[60] J. Tevis and J. Hamilton Jr. Static analysis of anomalies and security vulnerabilities in executable files. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 560–565, New York, NY, USA, 2006. ACM Press. 42

[61] H. Thompson, D. Beech, M. Maloney, and N. Mendelson. Xml schema part 1: Structures second edition. Technical Report REC-xmlschema-1-20041028, World Wide Web Consortium (W3C), 2004. 92, 94, 96

[62] Unknown. Executable and linkable format. http://www.x86.org/ftp/manuals/tools/elf.pdf. Last accessed on 9/24/2008. 20

[63] H. van Vliet. Mocha - the java decompiler. Available at http://www.brouhaha.com/~eric/software/mocha. Last accessed on 9/23/2008., 1996. 41

[64] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 264–279, Washington, DC, USA, 2006. IEEE Computer Society. 35

[65] W3C. Extensible markup language (xml). Available at http://www.w3.org/XML/. Last accessed on 9/23/2008., 2008. 10, 91

[66] M. Weber, V. Shah, and C. Ren. A Case Study in Detecting Software Security Vulnerabilities Using Constraint Optimization. In *SCAM*, pages 3–13, 2001. 28

[67] T. Welch. A Technique for High-Performance Data Compression. *IEEE Computer*, 17(6):8–19, 1984. 65

[68] D. Wheeler. Why Open Source Software/Free Software (OSS/FS,FLOSS, or FOSS)? http://www.dwheeler.com/oss_fs_why.html. Last accessed on 9/23/2008., 2007. 5

[69] T. Yetiser. Polymorphic Viruses, Implementation, Detection, and Protection. http://vx.netlux.org/lib/ayt01.html. Last accessed on 9/23/2008, 1993. 64