6-14-2012

Intra-procedural Path-insensitive Grams (i-grams) and Disassembly Based Features for Packer Tool Classification and Detection

Scott E. Gerics

INTRA-PROCEDURAL PATH-INSENSITIVE GRAMS (I-GRAMS) AND DISASSEMBLY BASED FEATURES FOR PACKER TOOL CLASSIFICATION AND DETECTION

THESIS

Scott E. Gerics, Captain, USAF

AFIT/GCE/ENG/12-07

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/12-07

INTRA-PROCEDURAL PATH-INSENSITIVE GRAMS (I-GRAMS) AND DISASSEMBLY BASED FEATURES FOR PACKER TOOL CLASSIFICATION AND DETECTION

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Engineering

Scott E. Gerics, B.S.C.E.

Captain, USAF

June 2012

AFIT/GCE/ENG/12-07

INTRA-PROCEDURAL PATH-INSENSITIVE GRAMS (I-GRAMS) AND DISASSEMBLY BASED FEATURES FOR PACKER TOOL CLASSIFICATION AND DETECTION

Scott E. Gerics, B.S.C.E.
Captain, USAF

Approved:

[illegible] 1 Jun 12

Rusty O. Baldwin, PhD (Chairman) Date

Thomas E Dube 1 JUN 12

Thomas E. Dube, PhD (Member) Date

William Kimball 1 Jun 12

William B. Kimball (Member) Date

AFIT/GCE/ENG/12-07

ABSTRACT

The DoD relies on computing devices to accomplish a wide range of goals and missions[1]. Malicious software jeopardizes these goals and missions. However, determining whether an arbitrary executable is malicious can be difficult. Obfuscation tools, called packers, are often used to hide malicious intent from anti-virus programs. Therefore detecting whether or not an untrusted executable file is packed is a critical step in software security.

This research uses machine learning methods to build the Polymorphic and Non-Polymorphic Packer Detection (PNPD) system that detects whether an executable is packed by either ASPack, UPX, Metasploit's polymorphic msfencode, or is packed in general. PNPD detection performance is evaluated on two feature sources. One is intra-procedural path-insensitive instruction sequences, referred to as i-grams. The other source, disassembly based features, consists of three sets: 1) control flow graph (CFG) structural information 2) control flow instructions (CFI), and 3) intermediate language (IL) representation of the Intel Architecture's x86 instruction set.

Both feature sources successfully detect packed executables used in experiments. Overall, it is discovered i-grams provide the best results with accuracies above 99.5%, average true positive rates above 0.977, and average false positive rates below 1.6e-3 when detecting msfencode packed executables. Grams of sizes 1 and 2 that exclude operands provide the best packed file detection results. The CFI feature set is the best performing disassembly source with performance results near that of the best i-grams.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Baldwin, for both his expert instruction and his balanced and consistent guidance. He provided the structure, knowledge, and freedom that kept me on track during the ups and downs I experienced and over the different paths I traversed. He made this relatively short journey a great learning experience. I would like to thank Mr. William Kimball for his advice, generous time, and astounding ability to approach problems in profound ways. His thought-provoking questions and ideas invigorated my mind and always moved my perspective far beyond the roadblocks. I would also like to thank Major Thomas Dube. His suggestions and constant support kept me motivated and moving during times of uncertainty. His passion for learning and the knowledge he shared led me to and down the final path of my research with excitement.

Finally, I would like to thank my wife and two children. The love, support, and sacrifices they made throughout this entire experience further brought to light how truly blessed I am to have them in my life.

Scott E. Gerics

TABLE OF CONTENTS

Page

ABSTRACT iv

ACKNOWLEDGMENTS v

LIST OF FIGURES ix

LIST OF TABLES xii

1 Introduction 1
- 1.1 Research Domain 1
- 1.2 Problem Statement 1
- 1.3 Research Goals 2
- 1.4 Document Outline 2

2 Background Information 3
- 2.1 Software Analysis Concepts 3
  - 2.1.1 Static Versus Dynamic Analysis 3
  - 2.1.2 Recursive Descent versus Linear Sweep Disassemblers 3
  - 2.1.3 Portable Executable Format 4
- 2.2 Malware and Packer Concepts 5
  - 2.2.1 Packers, Compressors, and Protectors 5
  - 2.2.2 Polymorphic versus Non-Polymorphic Packers 6
- 2.3 Malware and Packer Detection Research 7
  - 2.3.1 Pattern Based Malware Detection 7
    - 2.3.1.1 Control Flow Graph Based Signatures 8
    - 2.3.1.2 Static Analyzer for Executables 9
    - 2.3.1.3 Static Analyzer for Vicious Executables 9
  - 2.3.2 Machine Learning Based Methods 10
    - 2.3.2.1 Mining DLL Function Calls 10
    - 2.3.2.2 N-Gram Classifiers 11
    - 2.3.2.3 Malware Target Recognition 12
  - 2.3.3 Packer Detection 12
    - 2.3.3.1 PolyUnpack 12
    - 2.3.3.2 PE File Header Analysis 14
- 2.4 Opcode Based N-Grams 14
  - 2.4.1 Unknown Malcode Detection Using OPCODE Representation . . . 14
  - 2.4.2 Detecting Unknown Malicious Code by Applying Classification Techniques on OpCode patterns 15

2.5 Summary 16

3 Experimental Methodology 17
3.1 Background 17
3.2 Problem Definition 17
3.2.1 Goals and Hypotheses 17
3.2.2 Approach 19
3.3 System Boundaries 20
3.4 System Services 20
3.5 Workload 21
3.6 Performance Metrics 22
3.7 System Parameters 22
3.8 Factors 24
3.8.1 i-gram configuration factors 24
3.8.2 Disassembly based features configuration factors 25
3.8.3 i-gram and disassembly based features configuration factors 25
3.9 Class Sample Set Creation and Descriptions 25
3.9.1 Non-packed executables 25
3.9.2 Packed executables 26
3.9.2.1 UPX 26
3.9.2.2 ASPack 27
3.9.2.3 Polymorphic 27
3.9.2.4 Packed 28
3.10 Feature Set Description and Extraction 28
3.10.1 i-grams 28
3.10.2 CFG, CFI, and IL feature sets 31
3.10.2.1 CFG structural features 31
3.10.2.2 CFI features 31
3.10.2.3 IL features 31
3.11 Evaluation Technique 32
3.12 Experimental Design 32
3.13 Methodology Summary 33

4 Results and Analysis 34
4.1 I-gram Results and Analysis 34
4.1.1 Results across all i-gram configurations 34
4.1.1.1 Non-polymorphic classes 34
4.1.1.2 Polymorphic (msfencode) class 37
4.1.1.3 Packed class 40
4.1.2 Results for i-grams across classes independent of gram size and operand inclusion 42
4.1.3 Results for i-grams across gram sizes independent of class and operand inclusion 43

4.1.4 Results for i-grams across operand inclusion and exclusion configurations independent of gram size and class 45
4.1.5 Best and worst i-gram configurations tested 45
4.2 Disassembly Based Feature Sets Results and Analysis 48
4.2.1 Results across all disassembly-based feature configurations 48
4.2.1.1 Non-polymorphic classes 48
4.2.1.2 Polymorphic (msfencode) class 49
4.2.2 Packed class 51
4.2.3 Results for disassembly features across classes independent of feature set 52
4.2.4 Results for disassembly features across feature sets independent of class 53
4.3 Comparison of i-gram and structural and disassembly based features–What top feature sets are the best and worst for packer classification? 55
4.4 Summary of results and analysis 59

5 Conclusions 60
5.1 Research Accomplishments 60
5.2 Research Impact 60
5.3 Future Work 61
5.3.1 Test i-grams and disassembly based features at procedural level . . 61
5.3.2 Add inter-procedural support to i-grams 61
5.3.3 Time and resource performance improvement and analysis 61

APPENDIX A: All Confusion Matrices 63

APPENDIX B: Various Graphical Plots of Accuracy, TPR, FPR Results and Associated Mean Comparison Plots 66

APPENDIX C: Polymorphic Pilot Study Test Results for Polymorphic (msfencode) Class versus Non-Packed 79

BIBLIOGRAPHY 81

LIST OF FIGURES

FIGURE PAGE

2.1 Packer statistics (Feb '11 through Feb '12) [22] 6

2.2 Partial disassemblies for same executable packed by UPX (non-polymorphic) and msfencode (polymorphic) 7

3.1 Polymorphic and Non-Polymorphic Packer Detection (PNPD) System 21

3.2 Executable sample sets for all classification classes 27

3.3 Sample control flow graph 30

4.1 Comparison of information gains (computed by Weka) for 2-gram operands included and excluded configurations in polymorphic class 39

4.2 J48 decision tree (generated by Weka) for 1-gram operands excluded for packed class, all samples used 41

4.3 Comparison of accuracies for different i-gram sizes 44

4.4 Tukey HSD plot comparing different i-gram sizes by accuracy 44

4.5 Comparison of accuracies for the polymorphic and packed classes 46

4.6 Unique gram totals for i-gram configurations 47

4.7 Comparison of class accuracies 52

4.8 Comparison of class accuracies 53

4.9 Comparison of accuracies for feature sets 54

4.10 Comparison of true positive rates for feature sets 55

4.11 Comparison of false positive rates for feature sets 55

B.1 Accuracies for all i-gram configurations across all classes 66

B.2 True positive rates for all i-gram configurations across all classes 67

B.3 False positive rates for all i-gram configurations across all classes 67

B.4 Comparison of true positive rates for different i-gram sizes 68

B.5 Tukey HSD plot comparing different i-gram sizes by true positive rate 68

B.6 Comparison of false positive rates for different i-gram sizes 69
B.7 Tukey HSD plot comparing different i-gram sizes by false positive rate 69
B.8 Comparison of true positive rates for the polymorphic and packed classes . . . 70
B.9 Comparison of false positive rates for the polymorphic and packed classes . . . 70
B.10 Comparison of accuracies for disassembly based features 71
B.11 Comparison of true positive rates for disassembly based features 71
B.12 Comparison of false positive rates for disassembly based features 72
B.13 Tukey HSD comparison plot of UPX accuracies for disassembly based features 72
B.14 Tukey HSD comparison plot of UPX true positive rates for disassembly based features 73
B.15 Tukey HSD comparison plot of UPX false positive rates for disassembly based features 73
B.16 Tukey HSD comparison plot of ASPack accuracies for disassembly based features 74
B.17 Tukey HSD comparison plot of ASPack true positive rates for disassembly based features 74
B.18 Tukey HSD comparison plot of ASPack false positive rates for disassembly based features 75
B.19 Tukey HSD comparison plot of polymorphic accuracies for disassembly based features 75
B.20 Tukey HSD comparison plot of polymorphic true positive rates for disassembly based features 76
B.21 Tukey HSD comparison plot of polymorphic false positive rates for disassembly based features 76
B.22 Tukey HSD comparison plot of packed accuracies for disassembly based features 77

B.23 Tukey HSD comparison plot of packed false positive rates for disassembly based features 77

B.24 Tukey HSD comparison plot of packed true positive rates for disassembly based features 78

LIST OF TABLES

TABLE PAGE

3.1 Factors and factor levels for the PNPD system experiments 26

3.2 Sample sets used for experiments 29

4.1 Average accuracies, TPRs, and FPRs with confidence intervals for i-grams on UPX class 35

4.2 Average accuracies, TPRs, and FPRs with confidence intervals for i-grams on ASPack class 36

4.3 Average accuracies, TPRs, and FPRs with confidence intervals for i-grams on Poly class 37

4.4 Presence of top 2-grams (based on information gain for all samples) in polymorhpic (msfencode) class and non-polymorphic class executables 38

4.5 Presence of top 5-grams (based on information gain for all samples) in polymorhpic (msfencode) class and non-polymorphic class executables 38

4.6 Average accuracies, TPRs, and FPRs with confidence intervals for i-grams on Packed class 41

4.7 Average i-gram classification results for each class 42

4.8 Average i-gram results for operand inclusion 45

4.9 Average accuracies, TPRs, and FPRs with confidence intervals for disassembly features on UPX class 49

4.10 Average accuracies, TPRs, and FPRs with confidence intervals for disassembly features on ASPack class 50

4.11 Average accuracies, TPRs, and FPRs with confidence intervals for disassembly features on Poly class 50

4.12 Average accuracies, TPRs, and FPRs with confidence intervals for disassembly features on Packed class 51

4.13 Performance averages for top choices (All classes) 56
4.14 Performance averages for top choices (Polymorphic class) 57
4.15 Performance averages for top choices (Packed class) 57
4.16 Tukey HSD mean comparisons for accuracy (ACC), true positive rate (TPR), and false positive rate (FPR) for all classes 58
4.17 Tukey HSD mean comparisons for accuracy (ACC), true positive rate(TPR), and false positive rate(FPR) for polymorphic class 58
4.18 Tukey HSD mean comparisons for accuracy, true positive rate, and false alarm rate for packed class 58
A.1 Confusion matrix for all configurations based on 10-fold, cross-validation for all classes 63
C.1 Average accuracies, TPRs, and FPRs with confidence intervals for i-grams on poly pilot test 79
C.2 Confusion matrix for all i-gram configurations in poly pilot test based on 10-fold, cross-validation of msfencode executables versus non-packed executables) 80

INTRA-PROCEDURAL PATH-INSENSITIVE GRAMS (I-GRAMS) AND DISASSEMBLY BASED FEATURES FOR PACKER TOOL CLASSIFICATION AND DETECTION

1 Introduction

1.1 Research Domain

It is estimated over one billion personal computers were in use in 2008 [15]. These business, government, and individually owned computers are often targeted by criminals, terrorists, and enemy nation-states who use malware to confiscate the information, resources, or systems they contain or control. To mitigate this threat, widely used software security programs scan and detect executables with malicious intent. But at best, this is a cat and mouse game as malware creators constantly invent new methods to circumvent such detection systems employed. One such method is packing. The intent of packing is to compress or encrypt a malicious executable into a form unrecognized by software security programs. This leads to a false trust and the ultimate execution of the malware. To combat this, both static and dynamic based signature, heuristic, and machine learning methods are used. This research uses a static machine learning based method to detect the packed executables themselves.

1.2 Problem Statement

The USAF's and DoD's goals and missions rely heavily upon commercial software applications[1]. Therefore, ensuring a software application is not malicious is of critical importance to national security. To thwart detection of malicious applications, malware creators often resort to software obfuscation tools and techniques. Polymorphic and non-polymorphic packers are two such tools. Both types of packers alter the static appearance of an executable but maintain its runtime functionality. However, polymorphic type packers are more difficult to detect because they use mutation techniques to avoid signature-based packer detection tools. Because packing can bypass fast static malware detection methods and simple reverse engineering efforts, determining if an executable is packed is an important and necessary first step in determining whether to trust an arbitrary executable.

1.3 Research Goals

The goal of this research is to improve the software security methods available to the USAF and DoD. To accomplish this, the Polymorphic and Non-Polymorphic Packer Detection (PNPD) system uses machine learning with sequences of disassembled instructions and three distinct sets of disassembly based features to determine whether an executable is packed. The classification performance of these various feature sets is analyzed via a series of questions that culminate in determining candidates for the best and worst sets for detection.

1.4 Document Outline

Chapter 2 provides background information and relevant research related to this effort. Chapter 3 describes the experimental methodology used. Chapter 4 presents the results and analysis of the experiments conducted. Chapter 5 provides a summary and the conclusions for this research.

2 Background Information

This chapter provides an overview of both x86 based software concepts, malware and packer basics, and previous research in malware and packer detection.

2.1 Software Analysis Concepts

This section discusses concepts related to the structure and properties of windows x86 based software.

2.1.1 Static Versus Dynamic Analysis

Static and dynamic analysis are two primary ways of analyzing software and each have positive and negative characteristics. Static analysis examines the instructions of the binary executable under analysis but does not execute them. Dynamic analysis executes the instructions and observes the run-time behavior of the software on either real or virtual systems. Static analysis methods are commonly used for malware detection by antivirus programs because they often require less time to perform than dynamic analysis methods[25].

2.1.2 Recursive Descent versus Linear Sweep Disassemblers

Linear sweep and recursive descent are the two main types of disassembly algorithms [8]. The linear sweep disassembly algorithm assumes each machine code instruction is followed by another machine code instruction. Accordingly, the algorithm converts instructions sequentially relative to the first instruction. A major problem with this algorithm is it interprets any data in the code section of a program as an instruction. This leads to de-synchronization with respect to the actual instructions and inaccurate disassembly as many instructions subsequent to the data may be misinterpreted.

Recursive descent algorithms, also called traversal algorithms, such as those used by IDA Pro [8], use the entry point of a program as the starting point for disassembly and then disassemble instructions based on the program's control flow. For example, if a recursive descent algorithm interprets an instruction as a jump to an offset, the next instruction decoded is the target of the jump and disassembly continues from there until the next encountered branch instruction is reached and taken.

2.1.3 Portable Executable Format

Operating systems typically use some type of program header to provide information on how an executable program is organized, what external applications or libraries it references, and other attributes of the program to manage its execution. Windows based systems use the Portable Executable (PE) format to organize the various sections and other properties of executable files.

The PE format supports a variety of sections that represent different components of a program, however, not all are used and in many cases sections are combined. For this reason only the more common `.text`, `.data`, `.idata`, and `.rsrc` sections are discussed further.

The `.text` section usually contains all code generated by the program's compiler or assembler, one exception being the Borland C++ compiler which labels the section as `CODE` [16]. If the program is linked with multiple object files, the code from each of these is included in the `.text/CODE` section. The `.data` section contains runtime initialized data for the program's global and static variables and if multiple object files are linked, the section contains the combined `.data` sections of the files. Finally, the `.idata` section contains information pertaining to functions imported from DLLs distinct from the executable which, in many cases, can be most of the functionality of an application.

2.2 Malware and Packer Concepts

Most of the malware discovered “in the wild” is packed [22] to hide or obfuscate malicious behavior from reverse engineers, reduce the size of the packed code, and avoid static pattern matching antivirus detection applications. This section describes concepts relevant to malware, malware detection, and the techniques and tools used to pack it.

2.2.1 Packers, Compressors, and Protectors

Generally, packers are tools that allow a user to perform compression, encryption, bundling, or a combination of these on one or more executables or dynamic link libraries (DLLs) [9, 27]. A “compressor” is a packer that strictly attempts to compresses an executable and embeds a decompression algorithm that allows the compressed program to run. A “protector” packer obfuscates and protects the code within a program from detection by outside parties. Protectors typically accomplish protection through combinations of various compression, encryption, anti-disassembly, anti-debugging, and other advanced techniques such as using portable code machines. With respect to malware detection, packers transform the code and data contained in an executable file into an obfuscated form, encapsulated within another executable file. Figure 2.1 shows packer tools used between Feb 2011 and Feb 2012 to obfuscate collected malware samples[22]. Although the code of a packed executable appears changed on disk, packers typically use what are called packer stubs to maintain the code’s intended behavior at runtime. Thus, malicious software developers can develop one actual version of their malware and distribute alternate versions to evade signature-based anti-virus scanners [2]. Additionally, the functionality of such an application is protected from simple reverse engineering techniques and application programming interface (API) detection techniques since most packers hide the original import tables used by the malware.

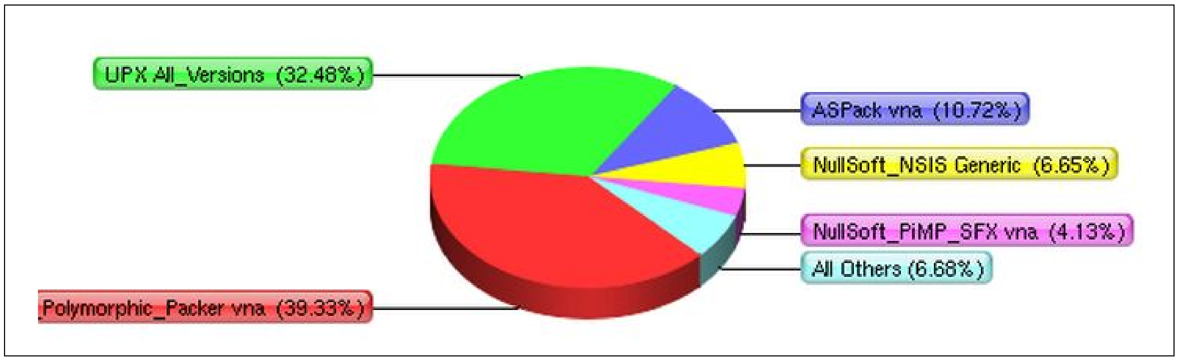
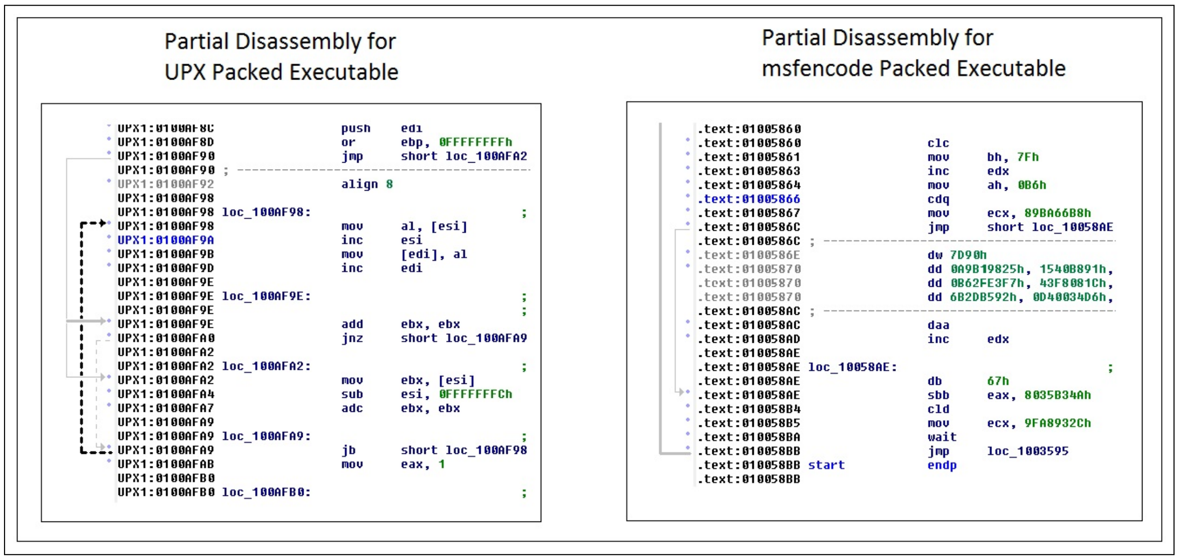


Figure 2.1: Packer statistics (Feb '11 through Feb '12) [22]

To pack and unpack a program, a packing algorithm transforms the executable code, data of interest, and relevant resources of an executable into an obfuscated and compressed state. Typically, a packing tool then inserts this transformed information into the program by altering the PE structure to accommodate this new information (data) with a packer stub. The packer stub is usually inserted within one of the new PE sections as executable code which allows the application or DLL to dynamically unpack and run the packed code during execution. A typical stub unpacks code by saving the current register state, performing decompression and or decryption on the target code, loading and linking original libraries required by the file, restoring the register state, and transferring execution to the original entry point (OEP) of the unpacked code. At this point the original, or slightly modified but functionally equivalent, code that was packed is now unpacked in memory.

2.2.2 Polymorphic versus Non-Polymorphic Packers

Polymorphic packers, such as msfencode, use encryption and mutation techniques to generate mutated versions of the same packed executable. Potentially millions of functionally equivalent, but statically altered, versions of the same executable can be created using polymorphism [25]. However, non-polymorphic packers, such as UPX, generate the exact same version of an executable each time is packed. In addition, non-polymorphic packers without protection methods use the exact or nearly exact sequences of instructions within the packer stubs they generate. Therefore simple signature-based detection of non-polymorphic packed executables is possible. Figure 2.2 shows partial disassemblies for the same executable packed with UPX and msfencode.



Partial Disassembly for
UPX Packed Executable

```
UPX1:0100AF8C                     push    edi
UPX1:0100AF8D                     or      ebp, 0FFFFFFFFh
UPX1:0100AF90                     jmp     short loc_100AFA2
UPX1:0100AF90 ; ---------------------------------------------
UPX1:0100AF92                     align 8
UPX1:0100AF98
UPX1:0100AF98 loc_100AF98:                                  ;
UPX1:0100AF98                     mov     al, [esi]
UPX1:0100AF9A                     inc     esi
UPX1:0100AF9B                     mov     [edi], al
UPX1:0100AF9D                     inc     edi
UPX1:0100AF9E
UPX1:0100AF9E loc_100AF9E:                                  ;
UPX1:0100AF9E                                               ;
UPX1:0100AF9E                     add     ebx, ebx
UPX1:0100AFA0                     jnz     short loc_100AFA9
UPX1:0100AFA2
UPX1:0100AFA2 loc_100AFA2:                                  ;
UPX1:0100AFA2                     mov     ebx, [esi]
UPX1:0100AFA4                     sub     esi, 0FFFFFFFCh
UPX1:0100AFA7                     adc     ebx, ebx
UPX1:0100AFA9
UPX1:0100AFA9 loc_100AFA9:                                  ;
UPX1:0100AFA9                     jb      short loc_100AF98
UPX1:0100AFAB                     mov     eax, 1
UPX1:0100AFB0
UPX1:0100AFB0 loc_100AFB0:                                  ;
```

Partial Disassembly for
msfencode Packed Executable

```
.text:01005860
.text:01005860                    clc
.text:01005861                    mov     bh, 7Fh
.text:01005863                    inc     edx
.text:01005864                    mov     ah, 0B6h
.text:01005866                    cdq
.text:01005867                    mov     ecx, 89BA66B8h
.text:0100586C                    jmp     short loc_10058AE
.text:0100586C ; ---------------------------------------------
.text:0100586E                    dw 7D90h
.text:01005870                    dd 0A9B19825h, 1540B891h,
.text:01005870                    dd 0B62FE3F7h, 43F8081Ch,
.text:01005870                    dd 6B2DB592h, 0D40034D6h,
.text:010058AC ; ---------------------------------------------
.text:010058AC                    daa
.text:010058AD                    inc     edx
.text:010058AE
.text:010058AE loc_10058AE:                                 ;
.text:010058AE                    db      67h
.text:010058AE                    sbb     eax, 8035B34Ah
.text:010058B4                    cld
.text:010058B5                    mov     ecx, 9FA8932Ch
.text:010058BA                    wait
.text:010058BB                    jmp     loc_1003595
.text:010058BB start              endp
.text:010058BB
```

Figure 2.2: Partial disassemblies for same executable packed by UPX (non-polymorphic) and msfencode (polymorphic)

2.3 Malware and Packer Detection Research

This section describes static malware and packer detection research efforts, and one hybrid approach to detect malware and packers.

2.3.1 Pattern Based Malware Detection

Most commercial malware detection systems use pattern matching as a technique to identify malicious code [3]. This approach searches for patterns of instructions or bytes in files to determine whether a unique signature of the malware is present. Although this syntactic analysis can accurately determine the specific type of malware via exact identification [25], it cannot typically detect new or modified versions of malware; simple obfuscation techniques can render a signature based scanner useless against a modified malware variant [2]. Therefore, many approaches that use pattern based detection methods attempt to de-obfuscate or interpret and define patterns semantically versus syntactically. This section describes a few of these efforts.

2.3.1.1 Control Flow Graph Based Signatures. Control flow signatures were used for worm detection in [12]. Control flow graphs of x86 based executable code discovered in network traffic was used to generate fingerprints to compare against code transmitted between computers on a network to determine whether the traffic exhibited worm-like behavior.

Using k-subgraphs, that is subgraphs of the overall control flow of the executable code under analysis, the system determines any isomorphic substructures between captured network traffic containing executable code. To determine the similarity of each k-subgraph, instructions within these structures are assigned a “color” to characterize the subgraph. This coloring technique ensures isomorphic k-subgraphs with different functionality are not considered equivalent. The efficiency of the comparisons is increased by using canonical graph labeling based on the Nauty library [13]. A worm is deemed present when matching fingerprints of executable code in network traffic is observed between a given number of distinct source-destination pairs, the traffic is observed on two or more internal networked machines, or the traffic was communicated between two or more hosts external to the network under analysis. The system is capable of identifying polymorphic worms, but due to its focus on the flow of network traffic versus the analysis of the executable code itself, high false positive rates occur when valid networked files are accessed and transmitted by users within the network.

2.3.1.2 Static Analyzer for Executables.

The Static Analyzer for Executables (SAFE) [4] distinguishes malware from benign code using static code and control flow abstraction. Although signature based, it is capable of detecting morphed versions of viruses due to a semantic interpretation of the signature. Generalized versions of both malicious and non-malicious executables are used. Specifically, abstract representations of known malicious executables in control flow form are stored in a database and used as abstraction patterns. This database is referenced by the system when an unknown executable is encountered to search for known malicious patterns within the executable's control flow graph representation. If malicious patterns are matched in the executable, a positive detection occurs. This particular effort resulted in the detection of viruses with zero false positives and negatives against the set of viruses tested.

2.3.1.3 Static Analyzer for Vicious Executables.

An effort to resolve detection issues in self-obfuscating malware, [24] proposed the Static Analyzer for Vicious Executables (SAVE) anti-virus scanner. Unlike SAFE, which uses control flow graphs, SAVE uses sequences of API calls used by known polymorphic malicious executables versus the executables themselves.

To determine whether an executable is malicious, the scanner decompresses the PE file (if compressed), parses the PE binary for the sequence of Windows API calls, and compares any sequence of API calls against a database of API calling sequence signatures which contains calling sequences from known variants of viruses that rely on the the Windows API, or Win32 viruses. API sequences are compared via a score-based sequence alignment algorithm that calculates a mean value using the Cosine measure [5], extended Jaccard measure [23], and Pearson correlation measures averaged together. The highest possible measure for each is one and therefore if the average measurement of a particular executable in question exceeds a 0.90 threshold value, it is considered malicious.

The SAVE scanner is implemented in [26] and detects all Mydoom, Bika, Beagle, and Blaster variants with perfect accuracy and outperforms the SAFE scanner based on time to scan Win32 PE files of various sizes.

2.3.2 Machine Learning Based Methods

Machine learning uses previously learned experiences to make decisions not possible with signature based methods alone. Unlike searching for specific patterns, heuristics use a historical approach and bases decisions on basic patterns from previous samples of identified malware and non-malware. This allows previously unseen malware to be detected. Furthermore, unlike signature based methods which require new signatures be added to an ever-growing collection, machine learning based solutions avoid significant amounts of storage overhead by using relatively small decision structures and classifiers built from training sets. However, machine learning based approaches have drawbacks as well. Two include the need to correctly identify and provide sufficient training samples as well as higher false positive rates. Several machine learning based malware detection schemes are discussed below.

2.3.2.1 Mining DLL Function Calls. Research performed by [18], uses a set of malicious and non-malicious MS-DOS based programs to train and test the accuracy of several data mining classifiers. The effort ultimately produced a detection rate of 97.76%. An up-to-date virus scanner differentiated between malicious and non-malicious executables for 4,266 files and it is assumed the scanner correctly identified each type. The two groups of executables are then split into training and test sets. Features extracted from the binary profiles of each executable consist of referenced DLLs, DLL function calls within the binary, the total number of function calls within the referenced DLLs, strings within the raw executable file, and byte strings obtained from hexadecimal editor dumps. The features are used to create RIPPER, Naïve Bayes, and Multi-Naïve Bayes classifiers.

The RIPPER algorithm is an inductive rule-based learner trained with the set of DLL function features extracted. The specific set of rules that define a malicious executable include no call to `user32.EndDialog()`, but a call to `kernel32.EnumCalendarInfoA()`; no call to `user32.LoadIconA()`, `kernel32.GetTempPathA()`, or any function in `advapi32.dll` DLL; one or more calls to `shell32.ExtractAssociatedIconA()`; one or more calls to functions in Microsoft Visual Basic Library DLL. If none of these conditions are met, the RIPPER algorithm deems the executable non-malicious. The Naïve Bayes classifier is trained with the sets of extracted strings and byte sequences. The multi-class classifier consists of six Naïve Bayes classifiers, each voting whether a particular unknown sample exhibits malicious behavior or not.

With regard to overall accuracy and detection rates, the Naïve Bayes algorithm and associated string features were the most accurate at 97.11% and produced the lowest false positive rate at 3.8%. The multi-Naïve Bayes and associated byte string features resulted in the second highest overall accuracy of 96.88% with false positive rates at 6.01%. The RIPPER algorithm and associated DLL features resulted in overall accuracies of 83.62% for the DLLs used feature set, 89.36% for DLL feature set, and 89.07% for DLLs with counted function calls.

2.3.2.2 N-Gram Classifiers. Similarly successful malware detection results were achieved by [11] using n-gram based feature extraction and classification techniques that produced a detector with 0.996 area under ROC curve.

Four-byte n-grams for 1,651 malicious executables and 1,971 non-malicious executables are created and consist of overlapping bytes that result in 255,904,403 total n-grams. As an example of 4-byte n-gram extraction, the following byte sequence of bytes, `0x3FA1C935779B`, would generate 3 unique n-grams consisting of `0x3FA1C935`, `0xA1C93577`, and `0x935779B`. These unique n-grams are sorted by their calculated information gain as $IG(j) = \sum_{v_j \in \{0,1\}} \sum_{C \in \{C_i\}} P(v_j, C_i) \log \frac{P(v_j, C_i)}{P(v_j)P(C_i)}$. Instance-based learner, Naïve Bayes, support vector machine, and decision tree classification methods are applied to the top 500 sorted n-grams per experimental run. In addition, a boosted support vector machine, Naïve Bayes, and decision tree are used as classifiers. The 0.996 area under ROC curve was achieved for the boosted decision tree classifier.

In addition to detection, malicious executables were also classified. Malicious executables are grouped into training and test sets for each class of malicious type. The same feature extraction and classification methods were used to produce results for three different types of malware: mass-mailer, backdoor, and executable virus with less successful results.

2.3.2.3 Malware Target Recognition. The Malware Target Recognition (MaTR) system [6] uses anomaly and structural features based on attributes of the PE structure of executables to determine whether a given executable is malicious. MaTR is contrasted against the performance of successful n-gram based research and is shown to outperform n-grams with statistical significance. In addition, MaTR uses only over 100 features and avoids the overhead n-grams incur with feature extraction and selection of grams from the millions of grams generated since the features extracted require little overhead to process. This effort yielded malware detection accuracies above 99.9%, false positive rates below 8.73e-4 and false negative rates below 8.03e-4.

2.3.3 Packer Detection

This section discusses research efforts that detect and or automatically unpack packed code.

2.3.3.1 PolyUnpack. [17] formally defines executables that unpack themselves prior to execution and additionally implements an automated un-packing tool.

The definition of an unpack-executing program is any program that executes instructions present at runtime in the text or data sections that are not decoded as the same instructions before the program is loaded. In addition, instructions determined to be executable in the code section, or those which were before the program was executed, also classify an executable as an unpack-executing program. This definition is used in the PolyUnpack tool which compares an executable before and after it is loaded to detect packed code and capture its unpacked representation.

To automatically unpack code, PolyUnpack statically analyzes an executable, executes it step-by-step and then compares the before execution and after execution instructions of the executable. If an instruction loaded in the program counter during the execution phase of detection cannot be found in any of the instructions from the static analysis of the executable, it is deemed an unpacked instruction. Executable code packed multiple times, i.e., packed code in packed code, is also detected by re-accomplishing the entire static-dynamic analysis process on each unpacked section of code up to a chosen number of iterations so that once one level of unpacking has finished, the system attempts another iteration of unpacking starting at the first instruction of the unpacked code previously discovered. Instructions within DLL calls are noted during the dynamic analysis phase but not analyzed.

Determining whether a program exhibits unpack-execute behavior is not decidable and reduces PolyUnpack's recognition to the Turing Halting problem. Therefore, the authors restrict the number of instructions executed during analysis of a program to a sufficiently low value.

PolyUnpack obtained generally better results than PEiD, a popular and free packer identifier, discovering 1,754 instances of packed executables versus 1,482 for PEiD, out of a total of 3,467 total samples. However, the number of actual packed executables in the overall sample set and whether the most recent PEiD signatures were used is unknown, leaving doubt as to the actual performance of the system.

2.3.3.2 PE File Header Analysis. The packed file detection technique developed by [20] detects packed executables based on features extracted from their PE headers. The characteristic vector of an eight element vector reveals variations that indicate a file is packed. The eight characteristic entities include values based on the number of executable and writable sections, the number of non-executable sections containing executable code, the number of unprintable section header names, whether there are any "no execute" sections, and whether the sum of all section sizes are larger than the file's overall size. These were chosen based on heuristic analysis performed on packed and non-packed files. One hundred packed samples were analyzed and the distances between the features compared to produce a minimum distance threshold to distinguish packed from unpacked files. Experimental results yielded 93.6% detection and 4% false positive rates.

2.4 Opcode Based N-Grams

2.4.1 Unknown Malcode Detection Using OPCODE Representation

Research conducted in [14] applies a new and successful method of n-gram based classification for malware detection using sample set of over 30,000 files. Instead of using the sequences of bytes contained within executable files, this research extracts sequences of opcodes, the operation portion of an instruction. These sequences are labeled as 'OpCodes'. The OpCodes are based on the disassembly provided by the IDA Pro disassembler. The authors state the sequences are created "in the same logical order in which the OpCodes appear in the executable" and are based on "execution flow of machine operations..." which implies control flow is respected, but a concrete example of extraction is not available.

The research centers around several questions regarding feature weighting, sequence size performance, top-selection method performance, classifier performance, and the classification imbalance problem. Term frequency and term frequency inverse document frequency are used for feature weighting comparisons. Individual sequence sizes from 1 to 6 were tested separately as features, versus combinations of sizes. The top 50, 100, 200, and 300 for along with document frequency, gain ratio and Fisher score are used to evaluate selection performance. Artificial Neural Networks, Decision Tree (unspecified version), Naïve Bayes, and the Adaboost.M1 versions were used for classifier performance comparisons. Finally, 5, 10, 15, 30, and 50 percent ratios of malware to non-malware are tested as classification training sets.

Performance results include an accuracy above 99% for OpCodes against a training set consisting of 15% malicious samples. The research results in best performance for sequences of size 2 versus other sizes and shows that larger sizes decrease accuracy. The Fisher score provides the best performance overall with the top 300 features and term frequency text categorization. The boosted decision tree, decision tree, and neural network classifiers provided the best classification performance.

2.4.2 Detecting Unknown Malicious Code by Applying Classification Techniques on OpCode patterns

A subsequent OpCode research effort conducted in [21] extends the results of [14]. The performance results for combined gram sizes is measured and how often a classifier should be trained is determined in addition to several questions from the previous work. The effort shows no performance gains when OpCodes of different sizes are combined into one feature set and reaffirms OpCodes of size 2 provide the best overall malware classification.

2.5 Summary

This chapter provides highlights on software analysis concepts, such as disassembly and the PE format used by windows executables. The function and techniques used by packer tools are described. In addition, prior work related to this research is discussed.

3 Experimental Methodology

3.1 Background

Determining whether an arbitrary program executable is malicious or not is a perennial problem. Malicious programs routinely steal usernames and passwords, confiscate personal information, and take control of machines [19] each month. Static and dynamic program analysis approaches assist in combating the propagation and use of such malicious programs. Of the two analysis methods, the static method is a popular approach because it analyzes programs in their pre or non-runtime states and avoids overhead dynamic methods incur with real or emulated execution of programs. Packing, however, can prevent static analysis by compressing, encrypting, or performing a combination of these two operations on compiled code. Thus, packing an executable potentially renders ineffective many static analysis techniques such as signature and heuristic based methods.

3.2 Problem Definition

This section describes the goals, hypothesis, and approach for this research.

3.2.1 Goals and Hypotheses

Packing an executable produces a statically different and often unrecognizable version of an executable through the compression and or encryption of its bytes. Because packing obfuscates the original bytes of an executable, simple static analysis methods may not recognize packed malicious applications. Packers typically use routines called packer stubs that decompress or decode packed code and data into program memory space when the program executes. Once unpacked in memory, the executable's original instructions and data are restored to their original state. Thus, if only static signature-based methods are employed for malware detection, a packed malicious application might execute undetected. Because the packer stub routines are typically the only code visible to static methods, signature-based detection tools often target stub instruction bytes. For non-polymorphic packers, such as UPX and ASPack, the packer stubs generated typically consist of similar instructions and control flow structures independent of the executable they pack. Therefore, signature-based detection of UPX and ASPack packed executables is fast and effective. However, polymorphic packers, such as Metasploit's msfencode tool, use engines that produce mutated versions of the same executable and packer stub each time the file is packed. Signature-based detection by virus scanners and packer detectors often miss such polymorphic packers.

This research evaluates polymorphic, non-polymorphic, and general packed detection performance of several disassembly based feature sets in conjunction with machine learning methods. These sets consist of two main sources. One uses intra-procedural path-insensitive instruction sequences (i-grams) with configuration options to include the size of the grams and whether the op-codes of the instructions or both op-codes and operands constitute the grams. The second source consists of three sets of generalized disassembly based features: a control flow graph (CFG) set, a control flow instructions (CFI) set, and an intermediate language (IL) set. A total of four classes representing two non-polymorphic based, one polymorphic based, and one general packed class are used. The general packed class provides classification results for packed versus not packed executables, similar to previous works that classify malicious from non-malicious executables. This research attempts to answer the following questions:

- Can i-grams be used to detect polymorphic and non-polymorphic packer tools?
- What packed classes do i-grams classify best?
- What gram sizes for i-grams perform the best?
- Do i-grams with or without operands provide better classification performance?

- What are the overall best and worst i-gram configuration candidates for packer classification?
- Can the CFG, CFI, and IL feature sets be used to detect polymorphic and non-polymorphic packer tools?
- What packed classes do the CFG, CFI, and IL feature sets classify best?
- What is the best choice for packer classification between CFG, CFI, and IL feature sets?
- What feature sets are the overall best and worst between the i-gram and CFG, CFI, and IL feature sets?

Based on previous research [14, 21], 2-grams are expected to perform the best for the i-gram group. It is expected that including operands with the grams will have a negative impact on polymorphic detection performance because they make each instruction in the gram more specific. In addition, classification results for the non-polymorphic classes should achieve the highest results because the disassembled code they produce is static compared to polymorphic packers. CFG, CFI, and IL feature sets are expected to classify and detect various packers fairly well based on pilot experiments, but no conjecture is made as to which one will perform the best. As a group, they are expected to perform better on the two UPX and ASPack non-polymorphic classes, similar to i-grams.

3.2.2 Approach

Static disassembly provides meaningful information about an arbitrary executable. Instructions and subsequent control flow information are used by reverse software engineers to understand the intent of an executable. This research uses disassembly information generated by the IDA Pro disassembler in conjunction with BinNavi, a control flow graph and intermediate language generation tool, to extract i-gram, CFG, CFI, and IL features from a large set of packed and non-packed executables. The packed sample set executables are created with several freely available packer tools. The non-packed executables are gathered from a fresh install of Windows XP service pack 2. Non-packed, polymorphic packed, and non-polymorphic packed type executables within this sample set provide both training and testing sets to build and test decision tree based classification. Classification results answer the performance questions poised in the previous section.

3.3 System Boundaries

The System Under Test (SUT) is the Polymorphic and Non-Polymorphic Packer Detection (PNPD) system. It consists of the components shown in Figure 3.1. The IDA Pro disassembler component converts the code and data sections of 32-bit Windows PE executable files, into Intel Architecture (IA) 32-bit instructions. BinNavi's control flow graph generator organizes the disassembled instructions by procedure into basic blocks with paths that describe the execution flow of the instructions and transforms IA instructions into a proprietary intermediate language. The feature extraction component extracts the i-gram, CFG, CFI, and IL sets of features from all procedures for each executable. Finally, the classification component uses the extracted features and associated classes to train and test a J48 decision tree classifier. The classification component is the component under test (CUT) and its classification output represents the SUT's output. This output is used to derive the accuracy, true positive rate, and false positive rate metrics used for performance evaluation. Finally, the SUT's workload consists of packed and non-packed executable files and their associated classes.

3.4 System Services

The PNPD system detects whether an executable has been packed by either a polymorphic or non-polymorphic based packer. The output of the system is a

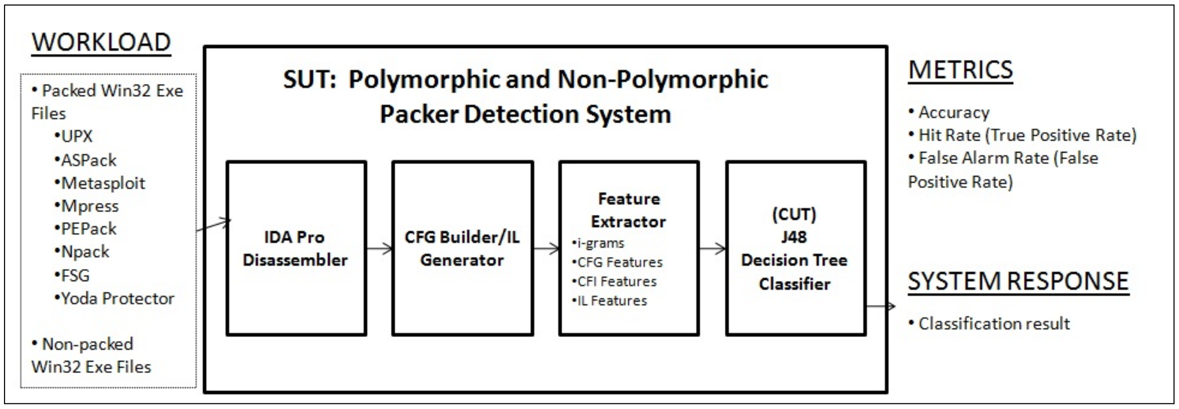


Figure 3.1: Polymorphic and Non-Polymorphic Packer Detection (PNPD) System

classification result that represents either a true positive, false positive, true negative, or false negative. System output depends on the classification class sets used to train the J48 decision tree component. This research refers to a true positive as a result that correctly identifies an executable within the classification class being detected. A false positive occurs when the classifier result incorrectly indicates an executable is not in the classification class when it actually is in the class. A true negative occurs when an executable not in the classified class is detected as such. Finally, a false negative occurs when an executable within the classified class is incorrectly determined as not in the class.

3.5 Workload

The workload for the PNPD system consists of packed and non-packed Win32 PE executables. It is restricted to Win32 x86-based executables due to the disassembly tools and packing software available for this format. Packed versions of windows system files are created using several freely available packers and constitute one set of workload input. These exercise the PNPD system's ability to detect both polymorphic and non-polymorphic packed files.

Workload parameters include whether a file is packed or not, the packer used to compress an executable, the version of the packer, configuration setting for the packer if applicable, and a class type associated with a file. These parameters directly affect the classification training and testing results, and therefore the overall classification performance of the system.

3.6 Performance Metrics

System performance is based on the ability of the system to accurately detect packed files as determined by the following:

True Positive Rate (TPR): A ratio of packed executables classified as such. This is calculated as the number of hits over the sum of the number of hits plus the number misses for the executable classified.

False Positive Rate (FPR): A ratio of non-packed executables incorrectly classified as packed. This is defined as the number of false positives over the sum of the false positives plus the true negatives for the executables classified.

Accuracy (ACC): A percentage of packed and non-packed executables correctly classified as such. This is defined as the sum of the true positives and the true negatives over the sum of the true positives, true negatives, false positives, and false negatives for the executables classified.

3.7 System Parameters

The following parameters affect the PNPD system's performance:

Decision Tree (DT) Type: The DT type affects the ability of the PNPD system to detect packed executables. Classifier type trees are available in different variations.

Decision Tree Configuration: The DT's configuration options affect classification accuracy. Various options are configurable and differ from one DT type to another.

Sample Sets Used to Train Decision Tree: The sample sets that train the DT can greatly impact classification performance results and must be chosen carefully to assure adequate training and meaningful testing. This ultimately determines how well the DT can resolve whether an executable is packed or not.

Packer Tools Used: Different packers or combinations of packers that are classified will lead to different classification performance results. Some packers generate packed files that are detected with greater accuracy versus other packed files

Obfuscation Techniques in Executables: Obfuscation techniques can hide features that would otherwise be extracted from the disassembled instructions and associated CFG of an executable. The feature set chosen must account for obfuscation techniques or assume obfuscation techniques are not used.

Type of Packing: Packers, in general, compress or encrypt executables or perform a combination of these functions. The function of the packer stub is coupled with the function of packer and therefore exhibits different characteristics. To build a robust DT, the type of packing must be accounted for with sufficient training samples.

Features Selection: The specific features chosen have a large impact on classification performance. However, due to the dimensional reduction inherent in DTs, all features chosen are used to build the DT classifier for packer stubs.

Number of Features: The number of features used to build a decision tree can negatively affect the DT's ability to accurately determine whether an executable contains a packer stub. If too few features are used, the DT runs the risk of not having enough features to split on, therefore reducing the accuracy of a DT determined result.

Algorithm Used for Feature Extraction: The algorithm that extracts features from the disassembled instructions and control flow graphs of executables affects the time required to process a request. Depending on the desired set of features, the algorithm may also impact the quality of information provided to the DT. For instance, if various forms and sequences of no-operation instructions is a sought after feature, the algorithm must understand the semantic equality between the `NOP` instruction and two `NOT AX` instructions used in succession.

Control Flow Graph Generator: Responsible for the input into the feature extraction component and therefore coupled with overall performance of the SUT.

Disassembler Type: Determines the way instructions are decoded. A recursive descent type disassembler decodes instructions by following the control flow of decoded instructions. A linear sweep type disassembler ignores control flow and decodes instructions assuming one instruction follows another. Recursive descent typically provides a more accurate representation of an executable's instructions [8].

3.8 Factors

Factors are split into three groups. One for the i-gram features, one for the disassembly based features, and another for both. These are summarized in Table 3.1.

3.8.1 i-gram configuration factors

The following i-gram configuration factors are tested:

i-gram size: The i-gram size is the number of instructions in the instruction sequence. For instance, an i-gram of size 2 would only consist of two sequential instructions. Sizes 1 through 5 are used for experimentation.

Operand inclusion: Operands can either be included or not included for i-gram sequences of instructions.

3.8.2 Disassembly based features configuration factors

The following disassembly based feature configuration factors are tested:

Feature set: The set of disassembly based features used for classification. Three sets are used. One consists of structural attributes of control flow graphs labeled as the CFG feature set. Another consists of the percentages of all IA x86 control flow instructions. The last consist of percentages based on IA x86 instructions translated into a proprietary set of intermediate language instructions.

3.8.3 i-gram and disassembly based features configuration factors

The following configuration factors are tested for both the i-gram and disassembly based features groups:

Classification class: This determines what class is classified. UPX, ASPack, Polymorphic, and Packed classes are used for experimentation.

3.9 Class Sample Set Creation and Descriptions

3.9.1 Non-packed executables

The non-packed executable files use for experiments consist of 623 Windows executables gathered from a fresh install of Windows XP SP2. Sizes for these files range from 1KB to 3.5MB with a median of 52KB and an average slightly above 153KB.

Table 3.1: Factors and factor levels for the PNPD system experiments

Group	Factor	Levels

i-grams	i-gram size
	Operand inclusion
		`1, 2, 3, 4, 5`
		`included, excluded`

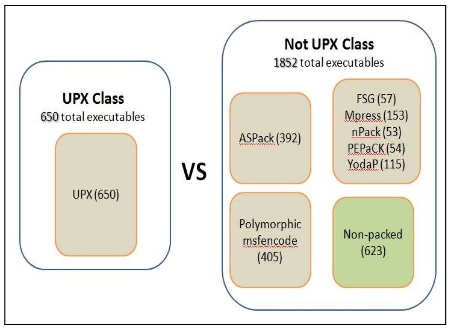
Disassembly based features	Feature set	`CFG set, CFI set, IL set`

Both i-gram and disassembly based features	Classification class	`UPX, ASPack,`
		`Polymorphic(msfencode),`
		`Packed`

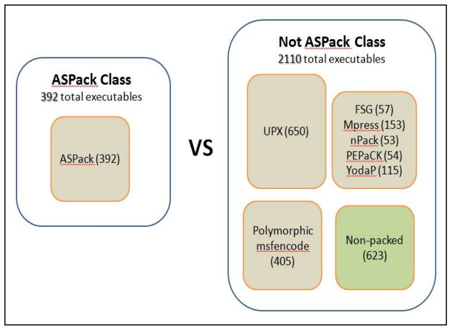
3.9.2 Packed executables

The packed executables used in the experiments are created by packing a subset of the non-packed Windows executables. A total of 1879 packed samples are used. Table 3.2 summarizes the various packers and versions used. The composition of each packed classification class sample set is discussed below and visible in figure 3.2.

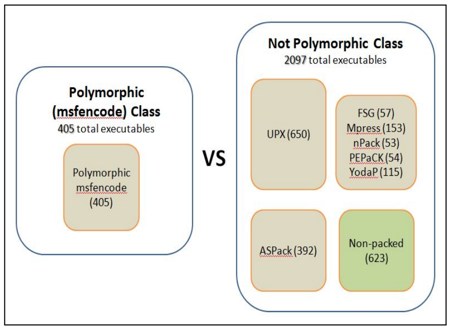
3.9.2.1 UPX. The 650 UPX packed executable files are created using four versions of UPX, specifically 1.2, 1.24, 2.03, and 3.08. Different configuration options are used for several versions, but preliminary investigation revealed that with one exception, neither changes in settings or version altered the features extracted from the compressed executable outputs. The notable exception for this is the LZMA option that is available in UPX version 3.08. Setting this option produced obvious differences in both the disassembled instructions and control flow graphs. For this reason, Table 3.2 only identifies the LZMA option for UPX. This class consists of the 650 UPX executables which are classified against the remaining 1229 packed and 623 non-packed executables from the sample pool.



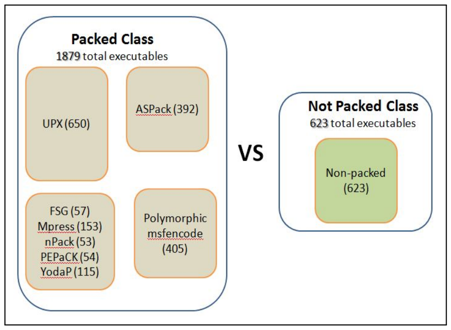
(a) UPX class samples



(b) ASPack class samples



(c) Poly class samples



(d) Packed class samples

Figure 3.2: Executable sample sets for all classification classes

3.9.2.2 ASPack. The 392 ASPack packed executable files are generated using seven different versions of ASPack, versions 2.0, 2.1, 2.11, 2.12, 2.2, 2.24, and 2.28. Preliminary investigation indicates that 2.12, 2.2, and 2.24 produce the same IDA Pro disassembly results consisting of two functions with only six instructions total between them. The samples for this class consist of the 392 ASPack executables which are classified against the remaining 1487 packed and 623 non-packed executables from the sample pool.

3.9.2.3 Polymorphic. The polymorphic packed executable files are generated using the msfencode tool included with Metasploit version 4.3.0. Specifically, msfencode embeds a reverse tcp payload into a total of 15 different non-packed executables with both the generic/none and x86/shikata_ga_nai encoders. Numerous iterations using the 15 files create the total 405 polymorphic packed samples used in experiments. The msfencode tool generates a different, hence polymorphically packed, executable for each iteration it outputs regardless of the encoder used. In other words, if the same executable, payload, and generic/none encoder are used three times, the resulting three different executables will have three different disassemblies generated. Therefore, the encoder chosen for msfencode seems irrelevant, but is still noted in the table below for completeness. This class consists of 450 msfencode executables which are classified against the remaining 1474 packed and 623 non-packed executables.

3.9.2.4 Packed. The Packed classification class represents a generalized packed class of both polymorphic and non-polymorphic packed samples. It is composed of all the packed samples from the UPX, ASPack, and Polymorphic classification classes with additional samples generated from several other packer tools. Specifically, the FSG, MPRESS, nPack, PEPaCK, and Yoda Protector tools. These additional tools do not use polymorphic techniques. As a result, the majority of samples for the packed set are of the non-polymorphic type. This class of 1879 packed executables is classified against the remaining set of all 623 non-packed executables.

3.10 Feature Set Description and Extraction

This section describes the i-gram, CFI, CFG, and IL features sets and the methods used to extract them.

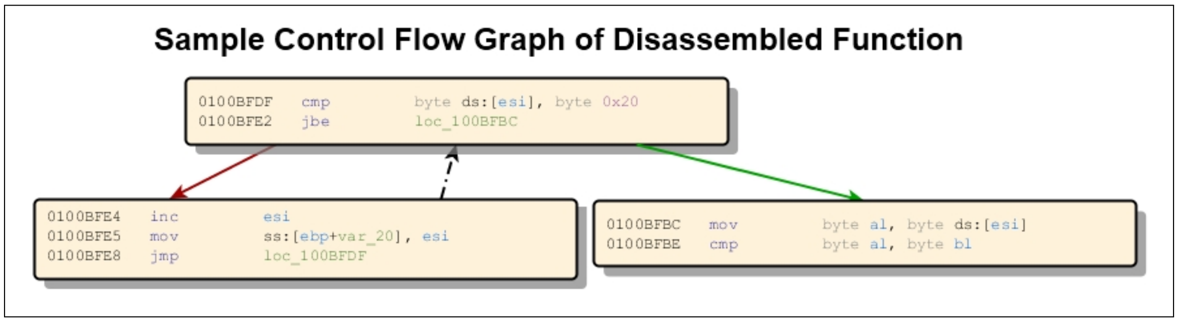
3.10.1 i-grams

For each executable used in the experiments, unique i-grams are extracted from every procedure produced by the IDA Pro disassembler and combined. i-Grams are represented as strings of instruction sequences extracted from the control flow graphs of the

Table 3.2: Sample sets used for experiments

Classification Class	Tool/Version/Config	Tool Sample Count
UPX	UPX/1.2	163
	UPX/1.24	56
	UPX/2.03	111
	UPX/3.08/LZMA Off	165
	UPX/3.08/LZMA On	155
ASPack	ASPack/2.0	55
	ASPack/2.1	55
	ASPack/2.11	55
	ASPack/2.12	55
	ASPack/2.2	55
	ASPack/2.24	55
	ASPack/2.28	62
Polymorphic	Metasploit msfencode/4.3.0/shikata	347
	Metasploit msfencode/4.3.0/none	58
Packed	ASPack/(prev. listed)	392
	FSG/2.0	57
	Metasploit msfencode/(prev. listed)	405
	Mpress/2.18	153
	nPack/2.0	53
	PEPaCK/1.0	54
	UPX/(prev. listed)	650
	Yoda Protector/1.0	115

procedures. Therefore, i-grams of size 2 or more incorporate instruction sequences that follow control flow while grams of size 1 do not. In addition, markers for nodes without entry or exit edges are also generated for i-grams of size 2 or greater. These are represented as a question mark character in the i-gram strings. To further illustrate this, consider the sample control flow graph generated for a procedure in Figure 3.3. Extraction for 1-grams without operands generates the following 5 unique i-grams: `(cmp)`, `(jbe)`, `(esi)`, `(mov)`, `(jmp)`. Extraction for 2-grams without operands generates 9 unique grams consisting of the following: `(?/cmp)`, `(cmp/jbe)`, `(jbe/inc)`, `(jbe/mov)`, `(inc/mov)`, `(mov/jmp)`, `(jmp/cmp)`, `(mov/cmp)`, `(cmp/?)`.



```
0100BFDF   cmp        byte ds:[esi], byte 0x20
0100BFE2   jbe        loc_100BFBC
```

```
0100BFE4   inc        esi
0100BFE5   mov        ss:[ebp+var_20], esi
0100BFE8   jmp        loc_100BFDF
```

```
0100BFBC   mov        byte al, byte ds:[esi]
0100BFBE   cmp        byte al, byte bl
```

Figure 3.3: Sample control flow graph

i-grams without operands use only the mnemonics, or opcodes, of instructions. i-grams with operands include the operands in the string representation of the i-gram. However, all numerical values within operands are generalized to a hash character for the string representation of the i-grams with operands. For example, 1-grams with operands consist of the following 7 unique i-gram strings for the sample figure: `(cmp;byte ds:[esi];byte #)`, `(jbe;loc_#)`, `(inc;esi)`, `(mov;ss:[ebp+var#];esi)`, `(jmp;loc_#)`, `(mov;byte al;byte ds:[esi])`, `(cmp;byte al;byte bl)`.

Once all unique i-grams are extracted from all executables, information gain, $IG(j) = \sum_{v_j \in \{0,1\}} \sum_{C \in \{C_i\}} P(v_j, C_i) \log \frac{P(v_j, C_i)}{P(v_j)P(C_i)}$ is applied to the unique grams based on what classification class is tested. The i-grams are sorted by information gain values and the top 500 are selected for classification use.

3.10.2 CFG, CFI, and IL feature sets

All features for the CFG, CFI, and IL sets use values based on percentages and averages. CFG features are generated from the structural attributes of control flow graphs for each executable. CFI features are created from the counts of control flow instructions in each executable. Finally, the IL set is based on Reverse Engineering Intermediate Language (REIL) instructions [7] generated by the BinNavi application for each disassembled executable. Further details on each feature set is discussed below.

3.10.2.1 CFG structural features. Their are 14 CFG structural features. All are averages or percentages relating to control flow based information such as the average number of instructions per block, average number of instructions per loop, and percentages of different types of edges.

3.10.2.2 CFI features. A total of 30 CFI features are generated by computing the percentage of all instructions each control flow instruction accounts for. As an example, if a total of 100 instructions are generated by the disassembly of an executable, and out of the 100 instructions 3 are `jnz` instructions, the `jnz` feature value is 3 percent.

Control flow instructions include all variants of the x86 jump, return, call, loop, and other control flow instructions. Percentages are chosen over the frequency of control flow instructions to provide more generalized values for classification.

3.10.2.3 IL features. The IL feature set consists of 22 REIL-based percentage values. Seventeen are the percentage of all REIL instructions in an executable that each REIL instruction accounts for. For example, a disassembled executable might contain the `and` instruction a total of 5 times out of a total of 100 instructions. A value of .05 is therefore calcalated for the `and` feature. The other 5 features are based on the arithmetic, bitwise, data transfer, logical, and “other” REIL categories of instruction functions. For example, an executable might consist of 10 percent arithmetic, 30 percent bitwise, 30 percent data transfer, 25 percent logical, and 5 percent other types of instructions functions.

3.11 Evaluation Technique

Direct measurement is used to assess the PNPD system. Both simulation and analytic techniques would require the design and implementation of systems more complex than a working system.

Various sets of packed and non-packed executable samples are processed using the Weka 3.6 machine learning software [10] to evaluate the PNPD system. The PNPD system, IDA Pro, BinNavi, and Weka run on a computer with 64-bit Windows 7 operating system, dual quad core 2.0 GHz processors, and 24GB RAM. IDA Pro 5.7 is used to disassemble executables. BinNavi 4.05 is used for control flow graph generation and provides the REIL instructions used as features. The PNPD system's output is the comma separated file generated by the Weka experimental environment. The file provides the tabulated classification results for each experiment run.

3.12 Experimental Design

A full factorial design is used to test the system. To ensure robust results, a stratified ten-fold cross-validation method creates the pseudo-random folds required for classifier training and testing.

For each experiment, the 2502 executables used in all experiments are divided into two sets. One set represents all executables in the class specified by the classification class factor level and the other set contains all remaining executables out of the total 2502 not in the selected classification class. Stratified 10-fold cross validation with random sampling is used. Each fold is tested against the Weka J48 decision tree implementation[10] trained by the other nine folds.

There are 5 size and 2 operand inclusion levels for the i-gram factors. The CFG, CFI, and IL sets represent the 3 feature set levels for the disassembly group. There are 4 classification class levels for both the i-gram and disassembly groups. Combined with the 10 cross-validation trials, a total of (5 i-gram sizes x 2 i-gram inclusion levels + 3 disassembly feature sets) x 4 classification classes x 10 folds = 520 experiments are generated.

With 20 replications, each generating new randomized folds, the total number of experiments is 520 x 20 = 10400. A 0.99 confidence level is used.

3.13 Methodology Summary

Obfuscation methods are often used to hide the malicious intent of malware from reverse engineers and software security applications. Polymorphic and non-polymorphic based packers transform the bytes and instructions of an executable into a form statically unrecognizable from their original form. This chapter outlines the packer detection goals of this research and the experimental approach and setup this research uses. Specifically, the PNPD system, which detects both polymorphic and non-polymorphic packing, is described. Operation is limited to x86 based 32-bit PE executables. A mixed set of both packed and non-packed Win32 executables with ten-fold cross-validation generates the training and test samples. Several common packers, including UPX, ASPack, and Metasploit's msfencode tool, generate the packed samples. A full-factorial design ensures all factor effects are observed.

4 Results and Analysis

This chapter begins with discussion of possible classification performance inflation due to the composition of sample sets used in experiments. It then answers several questions poised earlier regarding classification performance for the i-gram and disassembly features. Accuracy, true positive rate, and false positive rate metrics describe classification performance results. In addition, quantitative and qualitative observations describe the computational and resource requirements of the various i-gram and disassembly feature configurations. More emphasis is placed on results for msfencode polymorphic packed executable detection classification results as UPX and ASPack are easily detected using signature based methods.

4.1 I-gram Results and Analysis

This section discusses the results of all i-gram configurations.

4.1.1 Results across all i-gram configurations

This section discusses results across all i-gram configurations organized by class.

4.1.1.1 Non-polymorphic classes. The non-polymorphic UPX and ASPack classes result in good classification performance. UPX packed executables are detected with average accuracies at or above 99.76% and ASPack executables at or above 99.4% for all i-gram configurations. Further results and associated reasons are discussed further in this section.

Accuracy results in Table 4.1 show a general pattern of higher accuracy as gram size increases and when operands are included. UPX is detected with perfect accuracy at gram sizes of 2 or greater when operands are included and gram sizes of 3 or greater when operands are excluded. These results are expected. As gram size increases so does the

Table 4.1: Average accuracies, TPRs, and FPRs with confidence intervals for i-grams on UPX class

Configuration	Avg. ACC	99% CI	Avg. TPR	99% CI	Avg. FPR	99% CI
UPX/1 Gram/Excluded	99.76221	99.70604 – 99.81837	0.99854	0.99766 – 0.99942	0.00270	0.00197 – 0.00343
UPX/1 Gram/Included	99.96001	99.93789 – 99.98213	1.00000	1.00000 – 1.00000	0.00054	0.00024 – 0.00084
UPX/2 Gram/Excluded	99.95402	99.92726 – 99.98078	1.00000	1.00000 – 1.00000	0.00062	0.00026 – 0.00098
UPX/2 Gram/Included	100.00000	100.00000 – 100.00000	1.00000	1.00000 – 1.00000	0.00000	0.00000 – 0.00000
UPX/3 Gram/Excluded	100.00000	100.00000 – 100.00000	1.00000	1.00000 – 1.00000	0.00000	0.00000 – 0.00000
UPX/3 Gram/Included	100.00000	100.00000 – 100.00000	1.00000	1.00000 – 1.00000	0.00000	0.00000 – 0.00000
UPX/4 Gram/Excluded	100.00000	100.00000 – 100.00000	1.00000	1.00000 – 1.00000	0.00000	0.00000 – 0.00000
UPX/4 Gram/Included	100.00000	100.00000 – 100.00000	1.00000	1.00000 – 1.00000	0.00000	0.00000 – 0.00000
UPX/5 Gram/Excluded	100.00000	100.00000 – 100.00000	1.00000	1.00000 – 1.00000	0.00000	0.00000 – 0.00000
UPX/5 Gram/Included	100.00000	100.00000 – 100.00000	1.00000	1.00000 – 1.00000	0.00000	0.00000 – 0.00000

specificity of a gram's context of an executable. As an exaggerated example, consider a gram size of 100. Grams of this size extracted from the disassembled instructions of a particular executable are specific to that executable. Finding a similar gram consisting of the same sequence of 100 instructions in another executable is unlikely to occur unless the the same procedures are present in both executables. Such is the case, however, for UPX packed executables where the packer stub procedure used does not vary significantly or at all between files packed by UPX. Therefore, as gram size increases, they are more specific and identify longer sequences of instructions specific to only UPX packed executables. The other extreme is represented by 1-grams which are considered the most general of the gram sizes. They are defined by a single instruction and contain no control flow information and therefore provide the smallest context about an executable's disassembly. For example, a 1-gram consisting of the `call` instruction alone is far too general to discern between a UPX packed executable and any other arbitrary executable. Furthermore, the inclusion of operands increases specificity of grams. This is clearly evident within UPX results as 2-grams with operands included detect UPX with perfect accuracy. When operands are excluded, however, 2-grams are no longer specific enough to detect UPX with perfect accuracy.

Table 4.2: Average accuracies, TPRs, and FPRs with confidence intervals for i-grams on ASPack class

Configuration	Avg. ACC	99% CI	Avg. TPR	99% CI	Avg. FPR	99% CI
ASPACK/1 Gram/Excluded	99.40449	99.31668 – 99.49231	0.97170	0.96714 – 0.97626	0.00180	0.00123 – 0.00237
ASPACK/1 Gram/Included	99.83018	99.78334 – 99.87701	0.98980	0.98699 – 0.99262	0.00012	-1e-04 – 0.00034
ASPACK/2 Gram/Excluded	99.76819	99.70862 – 99.82776	0.98980	0.98699 – 0.99262	0.00085	0.00039 – 0.00131
ASPACK/2 Gram/Included	99.68029	99.61733 – 99.74324	0.98980	0.98699 – 0.99262	0.00190	0.00139 – 0.00241
ASPACK/3 Gram/Excluded	99.77023	99.71741 – 99.82305	0.98980	0.98699 – 0.99262	0.00083	5e-04 – 0.00116
ASPACK/3 Gram/Included	99.84015	99.79595 – 99.88436	0.98980	0.98699 – 0.99262	0.00000	0.00000 – 0.00000
ASPACK/4 Gram/Excluded	99.84015	99.79595 – 99.88436	0.98980	0.98699 – 0.99262	0.00000	0.00000 – 0.00000
ASPACK/4 Gram/Included	99.84015	99.79595 – 99.88436	0.98980	0.98699 – 0.99262	0.00000	0.00000 – 0.00000
ASPACK/5 Gram/Excluded	99.55833	99.48449 – 99.63217	0.98980	0.98699 – 0.99262	0.00334	0.00261 – 0.00408
ASPACK/5 Gram/Included	99.84015	99.79595 – 99.88436	0.98980	0.98699 – 0.99262	0.00000	0.00000 – 0.00000

ASPack detection results are shown in Table 4.1 and roughly follow the same pattern as those for UPX with notable differences for 5-grams without operands and 2-grams with operands configurations. Overall, the 1, 3, 4, and 5-gram with operands and 4-gram without operands configurations provide similar higher average accuracy results. The lowest average accuracy, approximately 99.83% occurs for the 1-gram without operands configuration. This accuracy is attributable to a relatively low true positive rate of 0.972 versus true positive rates of 0.99 and above for all other configurations. Results for 5-grams and 2-grams with operands configurations contrast the perfect classification results observed for UPX. The lower ASPack performance for these and 1-gram without operands configurations are likely due to the limited disassembly that IDA Pro produces for the 2.12, 2.2, and 2.24 versions of ASPack. IDA disassembly produces only two functions, one with a '`pusha, call`' sequence and the other with a '`pop ebp, inc` `ebp, push ebp, retn`' sequence. These short and general instruction sequences comprise nearly half of the ASPack samples used to build the classification training and test sets used for experimentation and negatively affects performance results.

4.1.1.2 Polymorphic (msfencode) class. Classification results for executables packed by Metasploit's msfencode tool are good with average accuracies above 97.0% for all configurations. Table 4.3 shows accuracies, true positive rates, and false positive rates for all i-gram configurations.

Table 4.3: Average accuracies, TPRs, and FPRs with confidence intervals for i-grams on Poly class

Configuration	Avg. ACC	99% CI	Avg. TPR	99% CI	Avg. FPR	99% CI
POLY/1 Gram/Excluded	99.19060	99.10481 – 99.27638	0.96865	0.96326 – 0.97403	0.00360	0.00285 – 0.00435
POLY/1 Gram/Included	99.37447	99.28708 – 99.46186	0.96137	0.95597 – 0.96678	0.00000	0.00000 – 0.00000
POLY/2 Gram/Excluded	99.50239	99.42084 – 99.58394	0.97745	0.97327 – 0.98163	0.00157	0.00086 – 0.00229
POLY/2 Gram/Included	99.56035	99.4874 – 99.6333	0.97783	0.97374 – 0.98191	0.00095	0.00058 – 0.00133
POLY/3 Gram/Excluded	99.30053	99.20256 – 99.39851	0.97350	0.96717 – 0.97982	0.00322	0.00236 – 0.00408
POLY/3 Gram/Included	99.17264	99.07558 – 99.2697	0.94891	0.94292 – 0.95491	0.00000	0.00000 – 0.00000
POLY/4 Gram/Excluded	99.14258	99.04809 – 99.23707	0.95273	0.94704 – 0.95842	0.00110	0.00067 – 0.00152
POLY/4 Gram/Included	98.47115	98.35151 – 98.59079	0.90559	0.89822 – 0.91295	0.00000	0.00000 – 0.00000
POLY/5 Gram/Excluded	98.22553	98.08166 – 98.3694	0.90426	0.89564 – 0.91287	0.00267	0.00203 – 0.00331
POLY/5 Gram/Included	97.00237	96.84484 – 97.15989	0.81662	0.80706 – 0.82618	0.00033	-2e-05 – 0.00069

A maximum average accuracy is observed at 2-gram configurations with diminishing accuracies as gram size increases from thereon. This contradicts the general increase in accuracy with increase in gram size observed for the non-polymorphic UPX and ASPack results due to the specificity of grams. As gram sizes increase, the specificity of context grows. For the non-polymorphic packed executables with similar or identical procedures, grams with greater sizes increase the information gained. This occurs because similar and longer instruction sequences common to all or most executables packed with a specific non-polymorphic packer are less likely present in executables not packed by the packer. However, grams of longer instruction sequences are not as common across all msfencode packed executables. Tables 4.4 and 4.5 affirm this. The top 2-grams are more common between all msfencode packed executables while the top 5-grams are less common. This results in lower information gain for higher sized grams concerning the polymorphic msfencode class and thus lower accuracies.

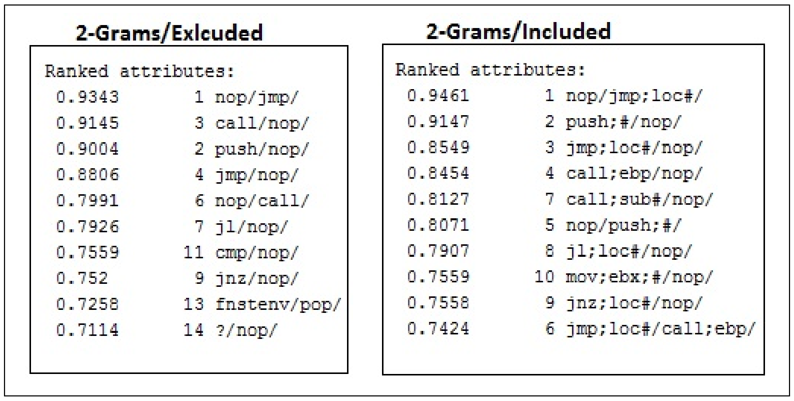
Table 4.4: Presence of top 2-grams (based on information gain for all samples) in polymorhpic (msfencode) class and non-polymorphic class executables

2-Gram Rank	2-Gram	Presence in Msfencode	Presence in Non-Msfencode
1	`(nop/jmp)`	396 of 405	4 of 2097
2	`(push/nop)`	395 of 405	10 of 2097
3	`(call/nop)`	390 of 405	4 of 2097
4	`(jump/nop)`	376 of 405	3 of 2097
5	`(nop/call)`	343 of 405	4 of 2097

Table 4.5: Presence of top 5-grams (based on information gain for all samples) in polymorhpic (msfencode) class and non-polymorphic class executables

5-Gram Rank	5-Gram	Presence in Msfencode	Presence in Non-Msfencode
1	`(nop/jmp/mov/nop/jmp/)`	238 of 405	0 of 2097
2	`(nop/jmp/call/nop/jmp/)`	193 of 405	0 of 2097
3	`(nop/jmp/push/nop/jmp/)`	176 of 405	0 of 2097
4	`(push/nop/jmp/push/nop/)`	174 of 405	0 of 2097
5	`(jmp/mov/nop/jmp/mov/)`	173 of 405	0 of 2097

The stratified 10-fold cross-validation method used for experiments also provides another explanation for the sharp drop in accuracy achieved by grams of size 5. Because there are fewer 5-grams common across all msfencode packed executables, training sets may not reflect actual information gain of a gram for the entire sample set. For example, the top 5-gram, `(nop/jmp/mov/nop/jmp/)`, for the polymorphic class is present in 238 out of the 405 msfencode packed executables and 0 of the 2097 non-msfencode packed executables. Stratified 10-fold cross-validation for the polymorphic set produces folds that contain roughly 40 msfencode packed executables and 209 non-msfencode packed executables. As an extreme example, random sampling fold generation might produce 5 folds that contain all 238 msfencode executables with the top 5-gram. If all 5 of these folds are present in the training set, the gram remains the top gram and therefore splits the J48 decision tree produced. However, because no msfencode packed executable in the test set contains the top gram, a greater number of false negatives, or lower true positives, are likely. This explains accuracy and true positive rates obtained using different gram configurations for the polymorphic class.



2-Grams/Exlcuded

```
Ranked attributes:
 0.9343       1 nop/jmp/
 0.9145       3 call/nop/
 0.9004       2 push/nop/
 0.8806       4 jmp/nop/
 0.7991       6 nop/call/
 0.7926       7 jl/nop/
 0.7559      11 cmp/nop/
 0.752        9 jnz/nop/
 0.7258      13 fnstenv/pop/
 0.7114      14 ?/nop/
```

2-Grams/Included

```
Ranked attributes:
 0.9461       1 nop/jmp;loc#/
 0.9147       2 push;#/nop/
 0.8549       3 jmp;loc#/nop/
 0.8454       4 call;ebp/nop/
 0.8127       7 call;sub#/nop/
 0.8071       5 nop/push;#/
 0.7907       8 jl;loc#/nop/
 0.7559      10 mov;ebx;#/nop/
 0.7558       9 jnz;loc#/nop/
 0.7424       6 jmp;loc#/call;ebp/
```

Figure 4.1: Comparison of information gains (computed by Weka) for 2-gram operands included and excluded configurations in polymorphic class

The average accuracy for the 1 and 2-gram with operands configurations are slightly higher than their counterpart without operand configurations. Although the differences are not statistically significant at the 0.99 confidence level, this difference is explained by the information gained with operands included. Figure 4.1 shows the differences of these information gains, calculated using Weka's information gain attribute evaluation method, between 2-grams with and without operands included. The 2-grams with operands included clearly shows higher information gains than those for operands excluded. This distinguishes msfencode's polymorphic engine against all other executables and reveals operands are not morphed across all instructions and instruction sequences for msfencode packed executables. However, accuracy for 3, 4, and 5-gram without operands decrease at a smaller rate than 3, 4, and 5-gram configurations with operands. Relatively sharp drops in accuracy for the 5-grams places them lower by 0.917% and 1.469% for operands excluded and included respectively. Grams become too specific and less common across msfencode packed executables for sequences of two instructions and including operands seems to multiply the reduction in performance (cf. Figures B.1.

4.1.1.3 Packed class. Performances observed for the packed class are lower overall than those for the UPX, ASPack, and polymorphic (msfencode) classes (cf. Figure 4.6). This seems intuitive based on the relative greater diversity of executables in the class. Specifically, the packed class consists of all 1879 packed executables and is classified against the 623 non-packed executables used in all experiments. Therefore, detection is generalized to a packed or not-packed result versus detection of a specific type of packer, as is the case for the UPX, ASPack, and polymorphic classes. Performance results show the 1-gram, operands excluded, configuration providing the significantly best performance for accuracy and false positive rate. Consequently, this suggests there are sets of common instructions across some packed executables that are not common across other non-packed executables, and vice-versa. This is evident based on the decision tree generated in Figure 4.2.

No other significant performance differences are observed between all other configurations. However, the steep drop in accuracy observed for the polymorphic class is not observed in results for the packed class, despite the presence of all msfencode packed

Table 4.6: Average accuracies, TPRs, and FPRs with confidence intervals for i-grams on Packed class

Configuration	Avg. ACC	99% CI	Avg. TPR	99% CI	Avg. FPR	99% CI
PACKED/1 Gram/Excluded	99.34656	99.24521 – 99.44791	0.99611	0.99524 – 0.99698	0.01454	0.01138 – 0.01769
PACKED/1 Gram/Included	98.61909	98.48942 – 98.74876	0.99566	0.99461 – 0.99672	0.04238	0.03756 – 0.04719
PACKED/2 Gram/Excluded	98.84499	98.72695 – 98.96302	0.99492	0.99396 – 0.99588	0.03106	0.02723 – 0.03488
PACKED/2 Gram/Included	98.65911	98.53173 – 98.78649	0.99867	0.99813 – 0.9992	0.04984	0.04481 – 0.05488
PACKED/3 Gram/Excluded	98.73705	98.61479 – 98.85932	0.99308	0.992 – 0.99416	0.02985	0.02574 – 0.03397
PACKED/3 Gram/Included	98.55716	98.41526 – 98.69906	0.99662	0.9959 – 0.99734	0.04776	0.04224 – 0.05327
PACKED/4 Gram/Excluded	99.09676	98.9907 – 99.20283	0.99811	0.99758 – 0.99864	0.03058	0.02656 – 0.0346
PACKED/4 Gram/Included	98.43926	98.30289 – 98.57563	0.99646	0.99566 – 0.99727	0.05201	0.04692 – 0.0571
PACKED/5 Gram/Excluded	98.97691	98.858 – 99.09582	0.99822	0.99756 – 0.99887	0.03571	0.03121 – 0.0402
PACKED/5 Gram/Included	98.40535	98.26249 – 98.5482	0.99715	0.99639 – 0.99791	0.05546	0.04994 – 0.06098

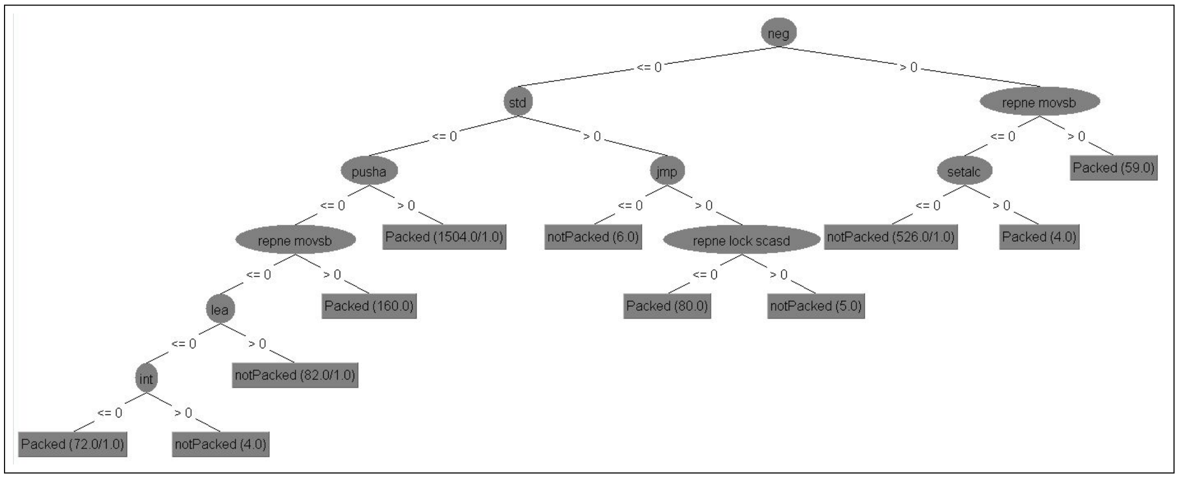


Figure 4.2: J48 decision tree (generated by Weka) for 1-gram operands excluded for packed class, all samples used

executables in the packed class. Once again, fold creation and information gain provide an explanation. The top 10 5-grams for both operand included and excluded configurations are present in most of the 623 non-packed executables and not present in most of the packed executables. Therefore, all folds generated have a higher probability of containing grams with high information gain and thus negative classification impacts incurred and previously discussed for the polymorphic class are not incurred by the packed class for 5-gram configurations.

4.1.2 Results for i-grams across classes independent of gram size and operand inclusion

Overall, classification results vary greatly based on the presence or absence of polymorphic packed executables. Performance details are discussed further.

Better i-gram performance for non-polymorphic packed executables is shown in Table 4.7. When the size and operand inclusion factors are ignored, UPX is classified with the highest average accuracy of 99.968% due to higher true positive rates and lower false positive rates than any other classified class. ASPack is detected with nearly the same performance of UPX with an average accuracy lower by a difference of 0.23 in percent, true positive rate lower by a difference of 0.011, and false positive rate higher by a difference of 4.9e-4. The low false positive rate indicates nearly perfect identification of non-ASPack executables.

Table 4.7: Average i-gram classification results for each class

Class	Accuracy	True Positive Rate	False Positive Rate
UPX	99.96762	0.99985	0.00039
ASPack	99.73723	0.98799	0.00088
Poly	98.89426	0.93869	0.00134
Packed	98.76822	0.99650	0.03892

Classification for msfencode and the classification of the combined packed set of UPX, ASPack, msfencode, yoda, mpress, npack, pepack, and fsg executables show significant differences from the UPX and ASPack classified sets. Polymorphic classification attains an average accuracy of 98.894% across folds. Packed classification attains a statistically significant lower average classification accuracy of 98.768%. The polymorphic performance is largely explained by the overall lower true positive rate, a result of polymorphic modules being misclassified as non-polymorphic. Although results for the packed class shows a true positive rate on par with the true positive rate for UPX, it suffers from a high false positive rate that is roughly 29 times that of the polymorphic and nearly a 100 times that of the UPX false positive rates. This explains the overall higher accuracy for polymorphic class detection.

4.1.3 Results for i-grams across gram sizes independent of class and operand inclusion

Overall, 2-grams perform the best closely followed by 1 and 3-grams. Poorer performance for 4-grams and a significantly poorer performance by 5-grams is observed. Results are discussed in detail below.

Two-grams attain the highest average accuracy, 99.496%, when classification class and operand inclusion factors are ignored as shown in Figure 4.3. The Tukey Honest Significant Difference (HSD) plot in Figure 4.4 shows 2-gram accuracies are not significantly different from 1 and 3-grams at the 0.99 confidence level, however, 2-gram true positive rates are the highest and do show significant difference from all other sizes (cf. Figures B.4 and B.5).

The lowest false positive rates are achieved by 1-grams, at 8.2e-3, with a statistically significant difference from 2-grams, at 1.08e-2 (Figures B.6 and B.7). The poorest false positive rate occurs with 5-grams at 1.22e-2. This suggests simpler i-grams are better at recognizing executables outside the classification class tested. Both 4 and 5-gram sizes produce accuracies less than and significantly different from 2-grams. As discussed

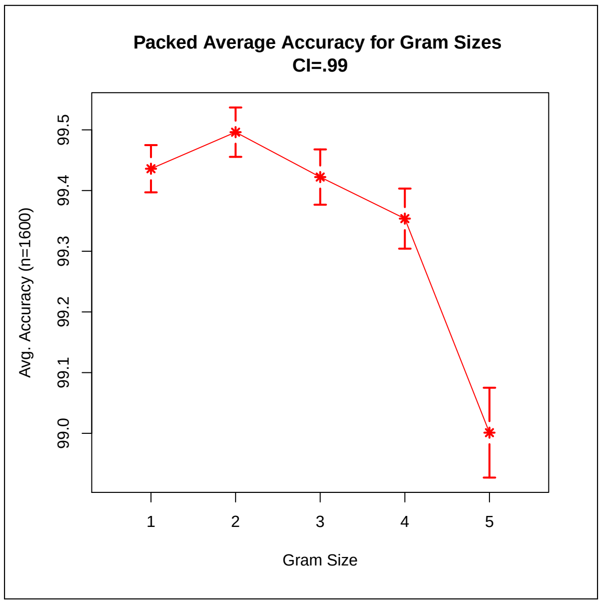


Figure 4.3: Comparison of accuracies for different i-gram sizes

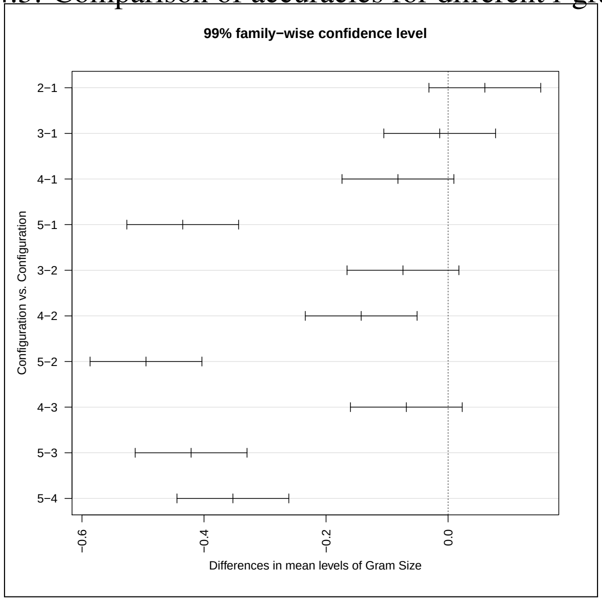


Figure 4.4: Tukey HSD plot comparing different i-gram sizes by accuracy

earlier, these gram sizes are poorer choices for detecting executables created by msfencode.

4.1.4 Results for i-grams across operand inclusion and exclusion configurations independent of gram size and class

Grams without operands perform better when gram size and class factors are not considered. Accuracy is 0.15849 higher in percent, true positive rate is higher by 7.12e-3, and false positive rate is lower by 4.36e-3 for exclusion versus inclusion of operands independent of class and gram size factors (cf. Table 4.8). The differences are all statistically significant at the 0.99 confidence level.

Table 4.8: Average i-gram results for operand inclusion

Operands	Accuracy	True Positive Rate	False Positive Rate
no	99.42108	0.98432	0.00820
yes	99.26259	0.97720	0.01256

As visible in previous figures and tables, grams with operands perform only slightly better for the UPX and ASPack classes. However, operand inclusion results in statistically significantly poorer classification for the polymorphic and combined classes when all gram sizes are considered. A statistically insignificant exception exists for the polymorphic class with the 1 and 2 gram sizes.

4.1.5 Best and worst i-gram configurations tested

The best and worst configurations are highly dependent on what is classified, but for generalized use, this research argues 1 or 2-grams without operands provide the best overall choice. Furthermore, any grams with sizes greater than 4 and any grams that include operands should be avoided.

Two-grams provide the most balanced results based on this research. As shown earlier, they provide the best classification results for the polymorphic class and also perform relatively well with the UPX and ASPack classes. However, 1-grams perform nearly as well over all the classes and require less computational overhead as discussed in the previous section. Furthermore, when polymorphic and combined packed performance results are combined, 1-grams are the most accurate(cf. Figures 4.5, B.8, and B.9). As observed earlier, two-grams without operands provide a statistically significant 0.091% gain in accuracy over 1-grams without operands for detection of polymorphic class executables. This supports the rationale of a qualitative tie between 1 and 2-gram without operands configurations.

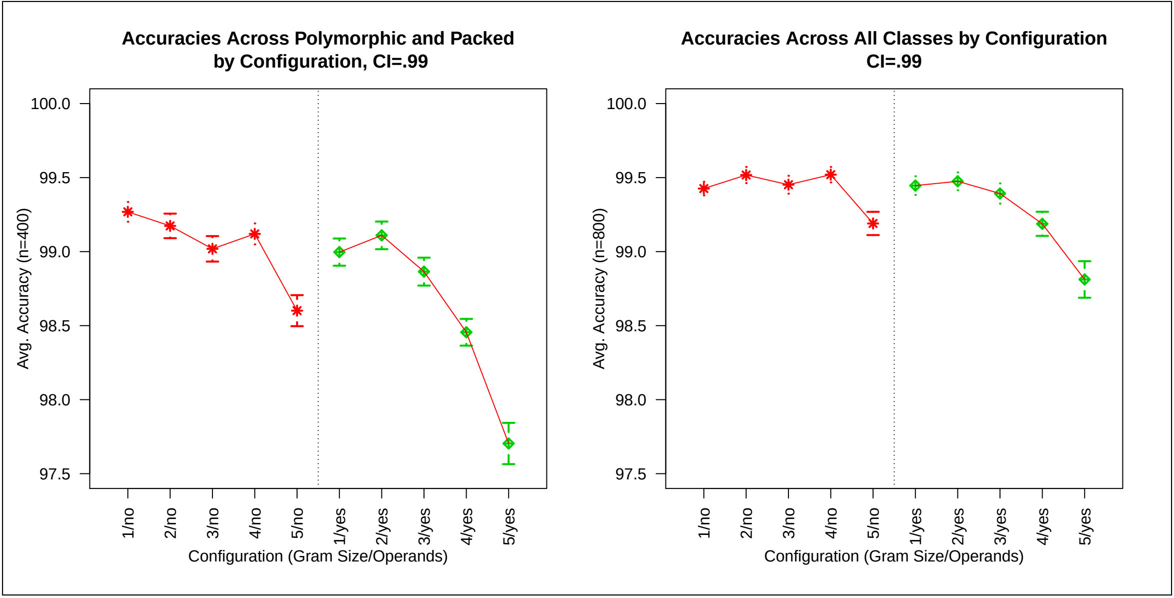


Figure 4.5: Comparison of accuracies for the polymorphic and packed classes

Operands, as created and tested for this research, are not worth including. The time required to compute and perform information gain calculations for i-grams that include operands require hours versus minutes to process for sizes greater than one. More information gain calculations and sorting is required due to the much larger number of unique grams generated (cf. Figure 4.6). Memory requirements obviously increase as well when operands are included. For these reasons, operands should be avoided in general when using i-gram information gain calculations.

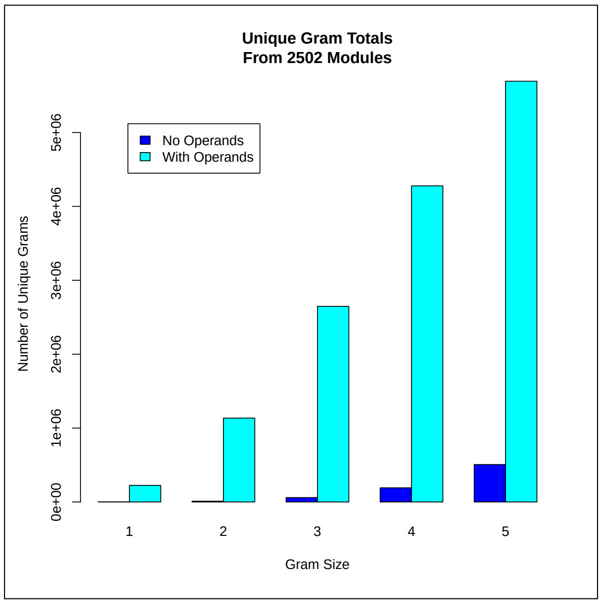


Figure 4.6: Unique gram totals for i-gram configurations

Grams larger than 4 should also be avoided. As seen in previous figures, there is a steep drop off in accuracy and true positive rate detecting the polymorphic class with grams of size 5 even when operands are excluded. Of course UPX and ASPack perform at or near perfect accuracy for these configurations. However, UPX and ASPack are easily detected using signature based packer scanners such as PEiD. Although 3 and 4-gram without operand configurations perform fairly well when results are observed across all classes, they are not considered as top choices due to the larger number of grams they generate.

In summary, the results of these i-gram experiments reveal that no particular i-gram configuration works best for all packers, but 1 and 2-grams without operands are effective for general and more practical purposes. Additionally, operands and grams with sizes greater than 4 should not be considered for classification use due to little or no detection benefit gained and higher resource cost incurred.

4.2 Disassembly Based Feature Sets Results and Analysis

This section discusses the performance results for the disassembly based CFG, CFI, and IL feature sets.

4.2.1 Results across all disassembly-based feature configurations

Results are discussed below for the non-polymorphic, polymorphic, and packed classes. CFG, CFI, and IL feature set results are shown in Tables B.10, B.11, and B.12. Consistent with i-gram results, UPX and ASPack classification results are higher than the polymorphic and packed classes. As discussed earlier, both UPX and ASPack generate highly static versions of packer stubs independent of the executable modules they compress. Therefore, the control flow graph, control flow instruction, and intermediate language attributes vary little between each module and thus provide more definitive classification. Disassembly based feature values for UPX and ASPack executables contrast well against other executables.

4.2.1.1 Non-polymorphic classes. The CFI set performs the best and the IL set performs the worst for UPX. The CFI feature set achieves the highest overall average accuracy, at 99.918%, highest true positive rate at 0.9984, and lowest false positive rate at 5.67e-4. Results for the CFI set are significantly different from the IL feature set with a 0.99 confidence level as shown in Figures B.13, B.14, and B.15. The IL feature set attains an average accuracy of 99.726%, true positive rate of 0.994, and false positive rate of 1.594e-3. CFI feature set performance is better than IL performance because IL features are more generalized than the CFI features. As with i-grams, UPX is detected better by features that contain more specific context. The CFI set consists of percentage values for all instructions within a disassembly for an executable. Because most of the UPX packed executables used in all experiments produce little or no variation between the disassemblies generated for them, higher specificity results in better classification.

Table 4.9: Average accuracies, TPRs, and FPRs with confidence intervals for disassembly features on UPX class

Configuration	Avg. ACC	99% CI	Avg. TPR	99% CI	Avg. FPR	99% CI
UPX/BasicCFG	99.78215	99.73805 – 99.82625	0.99438	0.99278 – 0.99599	0.00097	0.00056 – 0.00138
UPX/CF_Ins	99.91805	99.88567 – 99.95043	0.99846	0.99761 – 0.99931	0.00057	0.00025 – 0.00089
UPX/IL	99.72621	99.66792 – 99.7845	0.99400	0.99238 – 0.99562	0.00159	0.00103 – 0.00216

ASPack results show the CFG feature set performs just slightly better than the CFI set with average accuracy at 99.303% versus 99.293%, and true positive rate at 0.975 versus 0.9736 (cf. Table 4.10). However, the CFG set has a slightly higher average false positive rate at 3.63e-3 versus 3.48e-3. Similar to the UPX results, the IL set shows poorer overall performance than the other two sets for accuracy and true positive rate. The plots in , have means that are significantly different between sets. Based on TukeyHSD results (cf. Figures B.16, B.17 and B.18), CFI and IL set means are significantly different with regard to accuracies. For true positive rates, the IL set true positive rate average is significantly different from both the CFI and CFG sets. Finally, none of the sets show significant difference between their average false positive rates. Once again, a reasonable explanation for lower performance of the IL set is the more general nature of the IL set feature values. The IL set is more generalized than the CFI set because it converts a larger set of IA x86 instructions into a reduced set of only 17 possible instructions.

4.2.1.2 Polymorphic (msfencode) class. Results for the polymorphic class show better performance for IL features as shown in Table 4.11. Specifically, the IL set

Table 4.10: Average accuracies, TPRs, and FPRs with confidence intervals for disassembly features on ASPack class

Configuration	Avg. ACC	99% CI	Avg. TPR	99% CI	Avg. FPR	99% CI
ASPACK/BasicCFG	99.30252	99.19269 – 99.41234	0.97501	0.97027 – 0.97974	0.00363	0.00281 – 0.00444
ASPACK/CF_Ins	99.29263	99.20523 – 99.38002	0.97360	0.96915 – 0.97806	0.00348	0.00279 – 0.00417
ASPACK/IL	99.14663	99.05175 – 99.24151	0.96540	0.96028 – 0.97053	0.00370	0.0029 – 0.0045

performs best, CFI second, and CFG worst. The more general IL set of features attain a detection accuracy of 99.011% versus the CFG set with a statistically significant 98.721% (cf. Figure B.19). The IL set attains a higher true positive rate average of 0.9576 versus the CFI set, at 0.9504, and CFG set, at 0.9390 with statistically significant difference from the CFG set (cf. Figure B.20). All sets perform roughly the same with respect to false alarms with means between 3.48e-3 for CFG and 3.72e-3 for CFI and no significance in difference between any features sets (cf. Figure B.21).

Table 4.11: Average accuracies, TPRs, and FPRs with confidence intervals for disassembly features on Poly class

Configuration	Avg. ACC	99% CI	Avg. TPR	99% CI	Avg. FPR	99% CI
POLY/BasicCFG	98.72104	98.5911 – 98.85097	0.93902	0.93217 – 0.94588	0.00348	0.00273 – 0.00423
POLY/CF_Ins	98.88496	98.78034 – 98.98957	0.95042	0.94478 – 0.95606	0.00372	0.00283 – 0.00461
POLY/IL	99.01079	98.89832 – 99.12326	0.95758	0.95201 – 0.96314	0.00360	0.00275 – 0.00445

The IL performance suggests the instruction types (i.e. control flow, arithmetic, etc) that msfencode generates are somewhat consistent across iterations. The performance of CFI and CFG sets perform well and can characterize the polymorphic traits of msfencode. The performance of the CFG set suggests that the control flow graph structural features used have consistent values across iterations of msfencode. The same is true for the control flow instructions. Good detection results for the CFI set indicate control flow instruction percentage values for the polymorphic class contrast well against the values of executables not in the polymorphic class.

4.2.2 Packed class

As is the case for i-grams, detection results are lowest for the packed class (cf. Table 4.12. The average true positive rate for the packed class is higher than ASPack and polymorphic true positive rates. However, the packed set results are based on 1879 packed versus 623 non-packed modules which weights the false positive rate heavier in accuracy calculations than those for the UPX, ASPack, and polymorphic classes. In these other classes true positive rate has greater impact on accuracy. The CFG set provides the lowest detection rates for the packed class. It attains an average accuracy of 98.076%, a statistically significant lower accuracy than 98.667% and 98.521% for the CFI and IL features sets respectively (cf. Figure B.22). The CFG class results in an average false positive rate of 0.0472, significantly different from the CFI rate of 0.0296 and IL rate of 0.0343 (cf. Figure B.23). The higher false positive rate indicates more non-packed modules are misclassified as packed. Therefore, the CFG attributes of more non-packed modules are possibly not as discernible from the packed modules as those for the CFI and IL attributes.

Table 4.12: Average accuracies, TPRs, and FPRs with confidence intervals for disassembly features on Packed class

Configuration	Avg. ACC	99% CI	Avg. TPR	99% CI	Avg. FPR	99% CI
PACKED/BasicCFG	98.07561	97.91251 – 98.23871	0.98986	0.98853 – 0.99118	0.04725	0.04187 – 0.05262
PACKED/CF_Ins	98.66699	98.54207 – 98.7919	0.99195	0.99076 – 0.99314	0.02956	0.02582 – 0.0333
PACKED/IL	98.52114	98.39479 – 98.64749	0.99155	0.99034 – 0.99277	0.03429	0.03052 – 0.03805

A significant difference exists between the CFI, at 0.9916, and CFG, at 0.9899, true positive rates but not between these and the IL set, at 0.9915 (cf. Figure B.24). The true positive rate performance of the CFG attributes indicates that packed modules are missed by the classifiers more often than for the CFI and IL set based classifiers. Thus CFG feature values cannot discriminate packed from non-packed executables as effectively as the CFI and IL sets.

4.2.3 Results for disassembly features across classes independent of feature set

Classification results for the UPX, ASPack, polymorphic, and packed classes mimic results for i-grams (cf. Figure 4.7). UPX is detected with highest average accuracy, ASPack with second, the polymorphic class third, and finally the packed class fourth. All accuracies are significantly different from one another as shown in Figure 4.8.

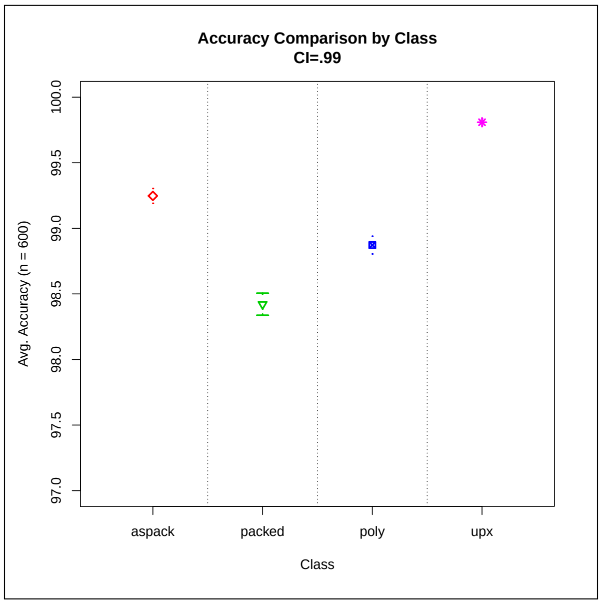


Figure 4.7: Comparison of class accuracies

Since UPX and ASPack tools produce "signature-like" code in files they compress, this provides better discrimination between the executables within the two classes versus those outside of them. UPX classification accuracy is higher than ASPack classification

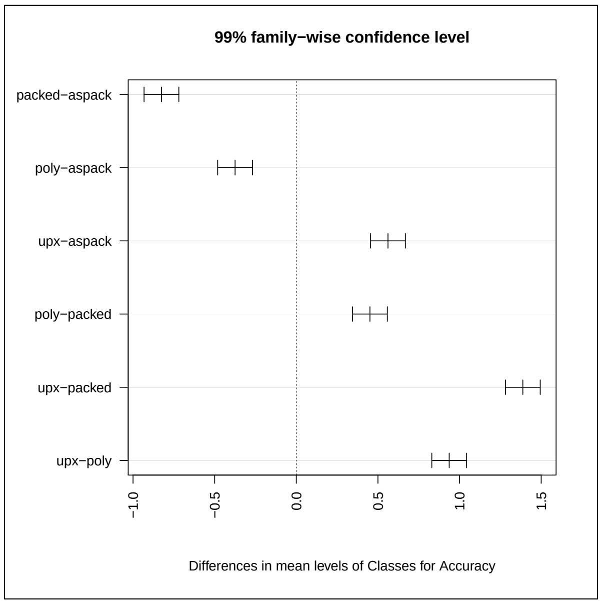


Figure 4.8: Comparison of class accuracies

accuracy because the UPX versions and settings used in the experiments produce virtually no change in the instructions disassembled or control flow graphs generated. Most ASPack versions, however, showed significant variation between their outputs. Polymorphic classification results top those for the packed class due to higher information gain attained by msfencode disassembly features versus those for the more diversified packed and non-packed classes.

4.2.4 Results for disassembly features across feature sets independent of class

The CFI feature set is the best choice of the three utilized based on a combination of its generally high performance and relatively modest computational costs. In contrast, the CFG feature set is the worst classifier due to its significantly worse performance across the polymorphic and packed classes.

When results are combined for all classes, the CFI set attains the highest accuracy and true positive rate and the lowest false positive rate (cf. Figures 4.9, 4.10, and 4.11). The CFI set also achieves the highest accuracy for the packed class, but the IL set attains the highest accuracy for the polymorphic class. However, both the difference between the

CFI and IL accuracies for polymorphic and packed classification are not significant at the 0.99 confidence level (cf. Figures B.19 and B.22). Computationally the CFI features require less time and fewer resources to compute than the IL set, which must convert every disassembled instruction into its intermediate form. For this reason and its overall better performance, the CFI set is the best choice between the three feature sets.

Regarding the worst choice, the CFG features provided the lowest performance relative to the other two sets for both the polymorphic and packed classes with significant differences at the 0.99 confidence level. In addition, the CFG features require computational resources above those for the CFI features. Even so, the features themselves could be considered for classification purposes. Although they are less effective than the other sets, they still provide an average accuracy near 99% and require somewhat less computational resources than the IL set. In addition, they might provide for better classification when combined with other features.

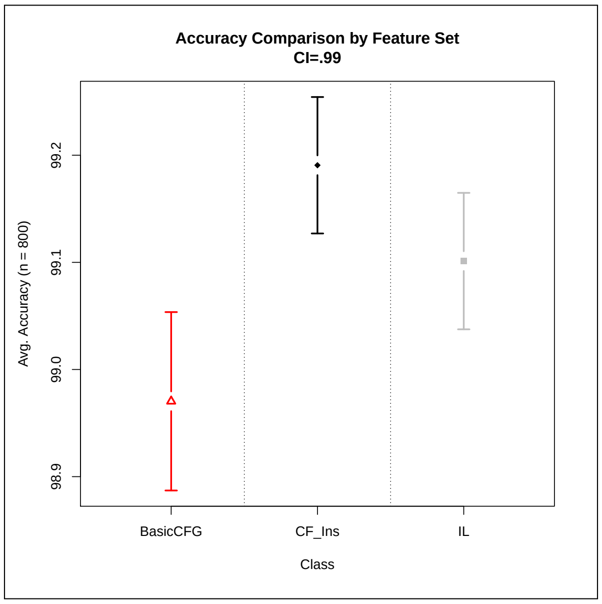


Figure 4.9: Comparison of accuracies for feature sets

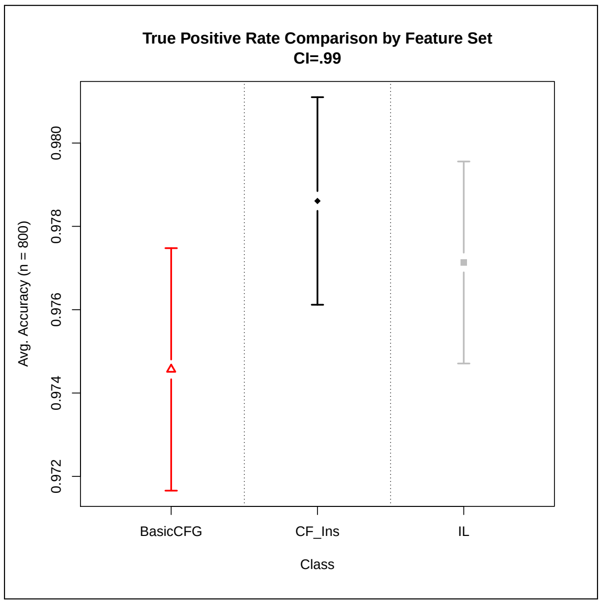


Figure 4.10: Comparison of true positive rates for feature sets

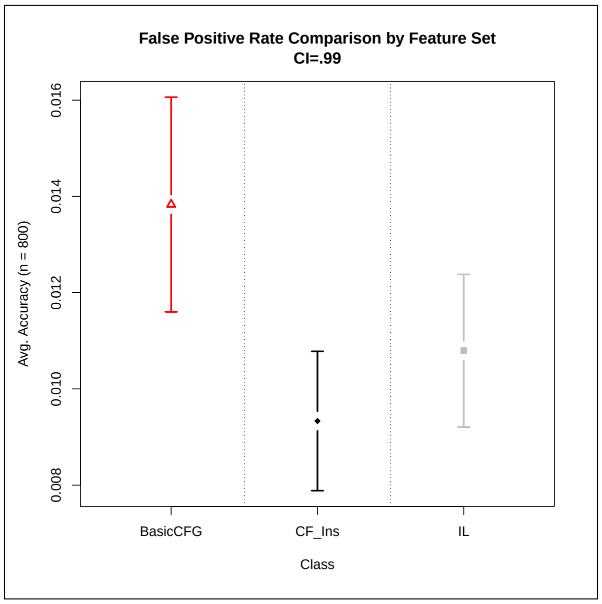


Figure 4.11: Comparison of false positive rates for feature sets

4.3 Comparison of i-gram and structural and disassembly based features–What top feature sets are the best and worst for packer classification?

Based on performance results and qualitative estimates of the computational resources required to extract features for each configuration, 1-grams, operands excluded, provide the best choice in general for packer classification compared to 2-grams and the CFI feature set. However, 2-grams should be considered for polymorphic purposes when the highest detection accuracy and true positive rate is desired. Finally, the CFI feature set shows promise but achieves the lowest performance between all three.

One-grams without operands provide the best overall effectiveness for cost. As mentioned in section previously, 1-grams without operands perform relatively well compared to the other i-gram configurations and require less computational resources than 2-grams without operands. In addition, they outperform the CFI attributes with statistical significance for all performance metrics across polymorphic, packed, and all classes combined, except for the polymorphic classification false positive rate (cf. Tables 4.13, 4.14, and 4.15). The Tukey HSD p-values in Tables 4.16, 4.17, and 4.18 indicate significant difference between the means at the 0.99 confidence level. The difference in computational resources between 1-grams and the CFI attributes are small because both require the inspection of all disassembled instructions within classified executables. Therefore, 1-grams are chosen as the best general purpose features for classification of packers.

Table 4.13: Performance averages for top choices (All classes)

Feature Set	Accuracy	True Positive Rate	False Positive Rate
2 Grm	99.51740	0.99054	0.00853
1 Grm	99.42596	0.98375	0.00566
CF_Ins	99.19066	0.97861	0.00933

Although 2-grams are more resource intensive than both 1-grams and the CFI attributes, they provide the highest polymorphic classification performance. Based on the

Table 4.14: Performance averages for top choices (Polymorphic class)

Feature Set	Accuracy	True Positive Rate	False Positive Rate
2 Grm	99.50239	0.97745	0.00157
1 Grm	99.19060	0.96865	0.00360
CF_Ins	98.88496	0.95042	0.00372

Table 4.15: Performance averages for top choices (Packed class)

Feature Set	Accuracy	True positive Rate	False positive rate
1 Grm	99.34656	0.99611	0.01454
2 Grm	98.84499	0.99492	0.03106
CF_Ins	98.66699	0.99195	0.02956

significantly higher average accuracy, higher true positive rate, and lower false positive rate than either 1-grams or the CFI features for the msfencode executables, 2-grams might also provide better discrimination for other polymorphic type tools. Furthermore, based on successful classification for the polymorphic class, 2-grams might provide classification for other purposes, such as identifying modules or procedures that provide similar functions but whose disassemblies are different.

The CFI features used in these experiments are a poorer choice than 1-grams for detecting polymorphic or packed classes, but show that simple statistics based on a relatively few number of control flow instructions can still provide good classification results. For this reason, more investigation into simple statistics based on executable disassemblies should be done.

Table 4.16: Tukey HSD mean comparisons for accuracy (ACC), true positive rate (TPR), and false positive rate (FPR) for all classes

Configurations Compared	p-val ACC	p-val HR	p-val FAR
2-Gram vs 1-Gram	0.00720	0.00000	0.00028
CFI vs 1-Gram	0.00000	0.00002	0.00000
CFI vs 2-Gram	0.00000	0.00000	0.51412

Table 4.17: Tukey HSD mean comparisons for accuracy (ACC), true positive rate(TPR), and false positive rate(FPR) for polymorphic class

Configurations Compared	p-val ACC	p-val HR	p-val FAR
2-Gram vs 1-Gram	0.00000	0.00456	0.00001
CFI vs 1-Gram	0.00000	0.00000	0.95870
CFI vs 2-Gram	0.00000	0.00000	0.00000

Table 4.18: Tukey HSD mean comparisons for accuracy, true positive rate, and false alarm rate for packed class

Configurations Compared	p-val ACC	p-val HR	p-val FAR
2-Gram vs 1-Gram	0.00000	0.07780	0.00000
CFI vs 1-Gram	0.00000	0.00000	0.00000
CFI vs 2-Gram	0.01287	0.00000	0.72377

4.4 Summary of results and analysis

This section argues both 1 and 2-grams without operands provide the best overall classification for the packers tested. Both perform well with average detection accuracies above 98.84% true positive rates above 0.969, and false positive rates below 0.031 across all four of the classes used in the experiments. In addition, both use fewer resources than i-grams of greater sizes and with operands. For polymorphic type detection purposes, 2-grams are better but 1-grams may provide better classification results for diversified classes. Simple disassembly based features also provide good packer detection results with average detection accuracies above 98.08%, true positive rates above 0.939, and false positive rates below 0.047.

5 Conclusions

5.1 Research Accomplishments

The results of this research answer several classification performance questions regarding i-grams and the CFG, CFI, and IL feature sets. I-grams classify and detect executables packed by both polymorphic and non-polymorphic packer tools with high accuracy, high hit rates, and low false alarm rates. Two-grams without operands provide an average accuracy above 99.5%, average hit rate above 0.977, and average false alarm rate below 1.6e-3 for Metasploit's msfencode executable output. Thus i-grams might detect other polymorphic tools as well if trained and tested. Regarding the best i-gram configuration, results both agree and disagree with previous results [14, 21]. Specifically, good performance for 2-grams without operands is observed, but 1-grams without operands also show good performance results and with fewer i-grams. Operand inclusion for i-grams leads to small performance gains over operand exclusion in some cases, but with much higher computational costs, adding millions more i-grams for gram sizes greater than 1.

Classification performance and computational requirements for the CFG, CFI, and IL sets indicate the CFI feature set is the best choice of the three. The best choice between the top two i-gram configurations and the CFI feature set are 1 and 2-grams based on statistically significant higher performance results and resource requirements.

5.2 Research Impact

This research demonstrates that machine learning with sequences of disassembled instructions and simple disassembly based features is successful for classification of the packed executables used for experiments within this research. These methods might prove useful for classification against larger sets of packed and non-packed executables. More importantly, these methods might provide useful solutions to other software security issues beyond the packer problem. Finally, this research expands upon previous instruction sequence related efforts. It confirms 2-grams as one of the best configurations for classification and tests the inclusion of generalized operands.

5.3 Future Work

5.3.1 Test i-grams and disassembly based features at procedural level

One technique employed by sophisticated malware creators involves deliberately not using obfuscation tools, but rather hiding malicious code in plain sight. Approaching problems at a more granular level, such as identification of procedures within an executable based on their functions, might provide high level information about an arbitrary executable useful for determining malicious or non-malicious intent. The i-gram and disassembly based features used in this research might provide the accurate classification necessary at this granular level.

5.3.2 Add inter-procedural support to i-grams

This research used i-grams based on and restricted to intra-procedural control flow. Expanding i-grams to include inter-procedural control flow information might yield higher performance results. This would require a more elaborate i-gram extraction procedure, as the order calls are invoked would have to be maintained.

5.3.3 Time and resource performance improvement and analysis

This research addresses the classification performance of i-grams and makes only obvious remarks regarding resources. The number of i-grams generated by each configuration was used to qualitatively compare and estimate the resources needed. However, if i-grams are to be used practically at an enterprise level, the time and resources required to extract, store, and use them should be measured. In addition, streamlining the processes required by i-grams should also be considered for practicality purposes.

5.3.4 Frequency based i-gram classification

This research limits the values of i-grams to either present or not present values. Incorporating frequency type information, similar to the averages and percentage values used for CFG, CFI, and IL features sets, might provide performance improvements for i-grams.

APPENDIX A: All Confusion Matrices

This appendix provides all confusion matrices based on average TP, FP, TN, and FN values for all experiments conducted by configuration. As discussed in the experimental methodology chapter, a TP occurs when the classifier correctly identifies an executable in the class specified by the configuration. A FP occurs when an executable not in the class specified by the configuration is classified as such. A TN occurs when an executable not in the class specified is classified as such. Finally, a FN occurs when an executable in the class specified is classified as an executable not in the class.

Table A.1: Confusion matrix for all configurations based on 10-fold, cross-validation for all classes

Configuration	TP	FP	TN	FN
ASPACK/1 Gram/Excluded	38.1	0.4	210.6	1.1
ASPACK/1 Gram/Included	38.8	0.0	211.0	0.4
ASPACK/2 Gram/Excluded	38.8	0.2	210.8	0.4
ASPACK/2 Gram/Included	38.8	0.4	210.6	0.4
ASPACK/3 Gram/Excluded	38.8	0.2	210.8	0.4
ASPACK/3 Gram/Included	38.8	0.0	211.0	0.4
ASPACK/4 Gram/Excluded	38.8	0.0	211.0	0.4
ASPACK/4 Gram/Included	38.8	0.0	211.0	0.4
ASPACK/5 Gram/Excluded	38.8	0.7	210.3	0.4
ASPACK/5 Gram/Included	38.8	0.0	211.0	0.4
ASPACK/BasicCFG	38.2	0.8	210.2	1.0
ASPACK/CF_Ins	38.2	0.7	210.3	1.0
ASPACK/IL	37.8	0.8	210.2	1.4
PACKED/1 Gram/Excluded	187.2	0.9	61.4	0.7

PACKED/1 Gram/Included	187.1	2.6	59.7	0.8
PACKED/2 Gram/Excluded	186.9	1.9	60.4	1.0
PACKED/2 Gram/Included	187.7	3.1	59.2	0.2
PACKED/3 Gram/Excluded	186.6	1.9	60.4	1.3
PACKED/3 Gram/Included	187.3	3.0	59.3	0.6
PACKED/4 Gram/Excluded	187.5	1.9	60.4	0.4
PACKED/4 Gram/Included	187.2	3.2	59.1	0.7
PACKED/5 Gram/Excluded	187.6	2.2	60.1	0.3
PACKED/5 Gram/Included	187.4	3.5	58.8	0.5
PACKED/BasicCFG	186.9	2.9	58.5	1.9
PACKED/CF_Ins	187.3	1.8	59.6	1.5
PACKED/IL	187.2	2.1	59.3	1.6
POLY/1 Gram/Excluded	39.2	0.8	208.9	1.3
POLY/1 Gram/Included	38.9	0.0	209.7	1.6
POLY/2 Gram/Excluded	39.6	0.3	209.4	0.9
POLY/2 Gram/Included	39.6	0.2	209.5	0.9
POLY/3 Gram/Excluded	39.4	0.7	209.0	1.1
POLY/3 Gram/Included	38.4	0.0	209.7	2.1
POLY/4 Gram/Excluded	38.6	0.2	209.5	1.9
POLY/4 Gram/Included	36.7	0.0	209.7	3.8
POLY/5 Gram/Excluded	36.6	0.6	209.1	3.9
POLY/5 Gram/Included	33.1	0.1	209.6	7.4
POLY/BasicCFG	38.0	0.7	209.0	2.5
POLY/CF_Ins	38.5	0.8	208.9	2.0
POLY/IL	38.8	0.8	208.9	1.7
UPX/1 Gram/Excluded	64.9	0.5	184.7	0.1

UPX/1 Gram/Included	65.0	0.1	185.1	0.0
UPX/2 Gram/Excluded	65.0	0.1	185.1	0.0
UPX/2 Gram/Included	65.0	0.0	185.2	0.0
UPX/3 Gram/Excluded	65.0	0.0	185.2	0.0
UPX/3 Gram/Included	65.0	0.0	185.2	0.0
UPX/4 Gram/Excluded	65.0	0.0	185.2	0.0
UPX/4 Gram/Included	65.0	0.0	185.2	0.0
UPX/5 Gram/Excluded	65.0	0.0	185.2	0.0
UPX/5 Gram/Included	65.0	0.0	185.2	0.0
UPX/BasicCFG	64.6	0.2	185.0	0.4
UPX/CF_Ins	64.9	0.1	185.1	0.1
UPX/IL	64.6	0.3	184.9	0.4

APPENDIX B: Various Graphical Plots of Accuracy, TPR, FPR Results and Associated Mean Comparison Plots

This appendix provides graphical views of results that complement discussion and tables listed in the results and analysis chapter.

B.0.5 Accuracy, TPR, and FPR Plots for All i-gram Configurations

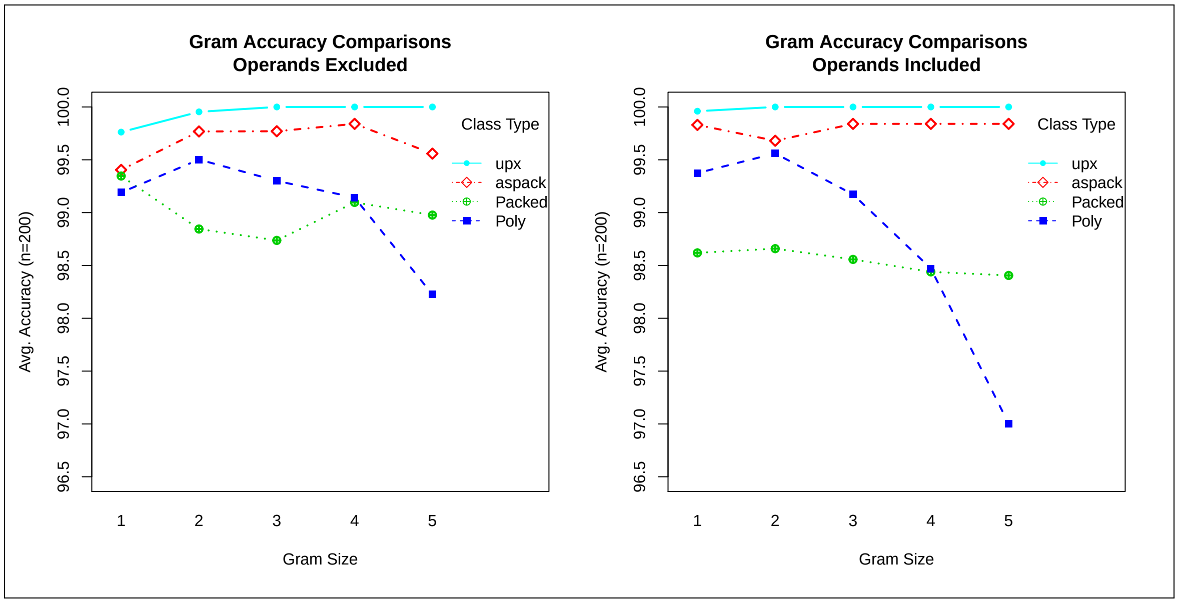


Figure B.1: Accuracies for all i-gram configurations across all classes

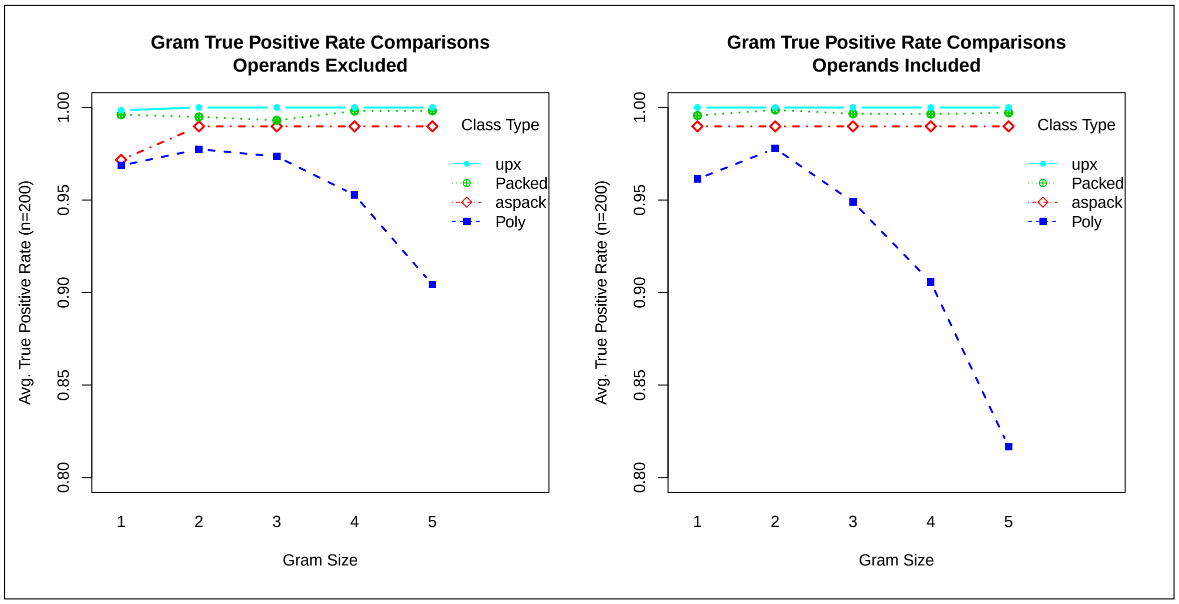


Figure B.2: True positive rates for all i-gram configurations across all classes

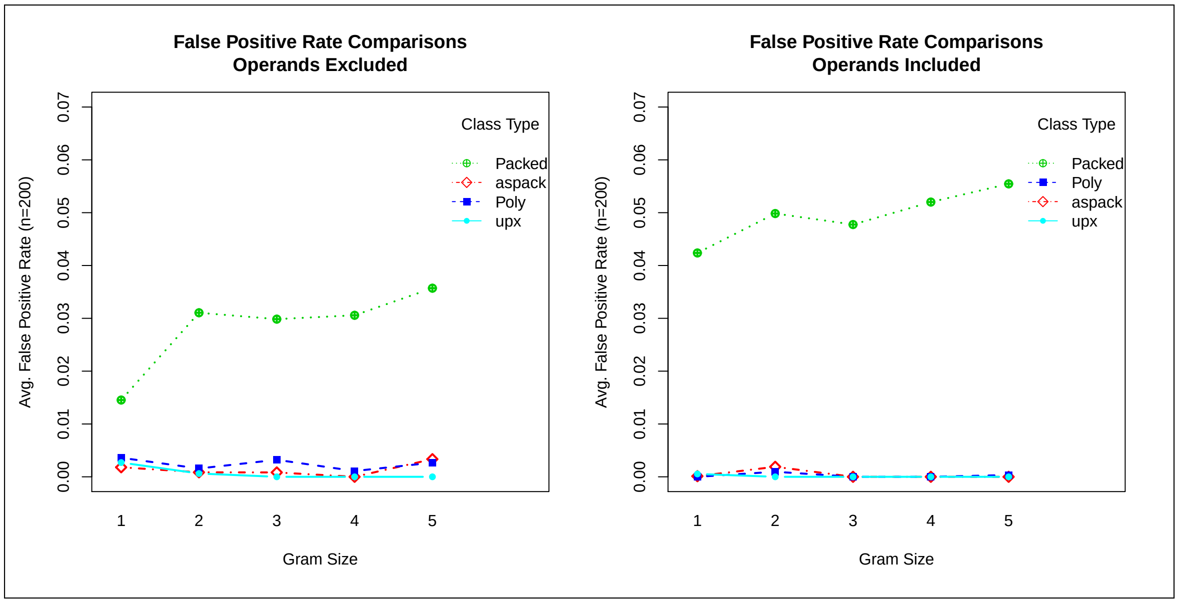


Figure B.3: False positive rates for all i-gram configurations across all classes

B.1 TPR, FPR, and Associated Tukey HSD Plots for i-gram Sizes

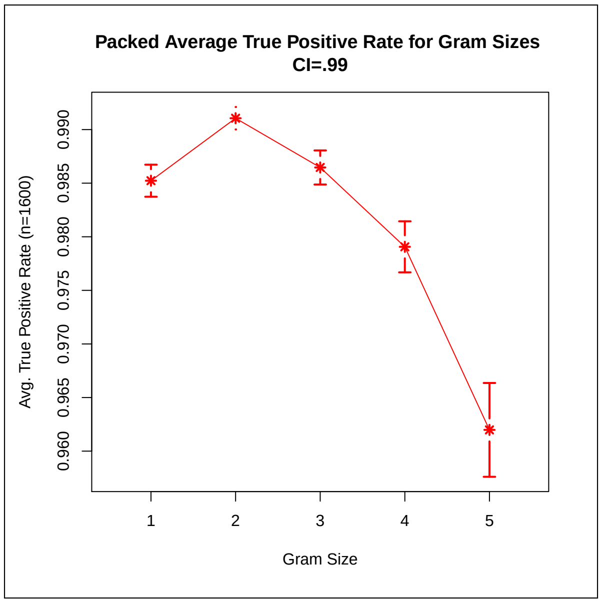


Figure B.4: Comparison of true positive rates for different i-gram sizes

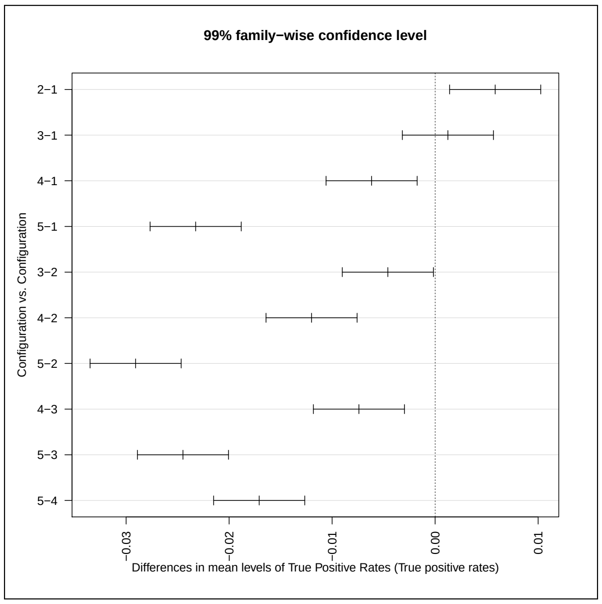


Figure B.5: Tukey HSD plot comparing different i-gram sizes by true positive rate

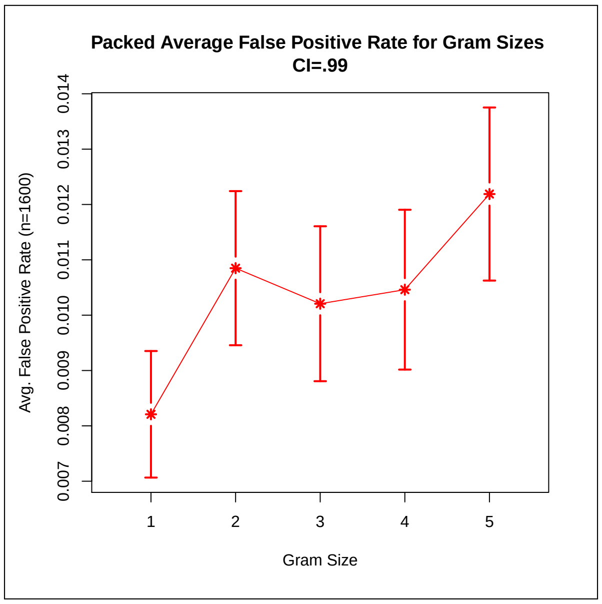


Figure B.6: Comparison of false positive rates for different i-gram sizes

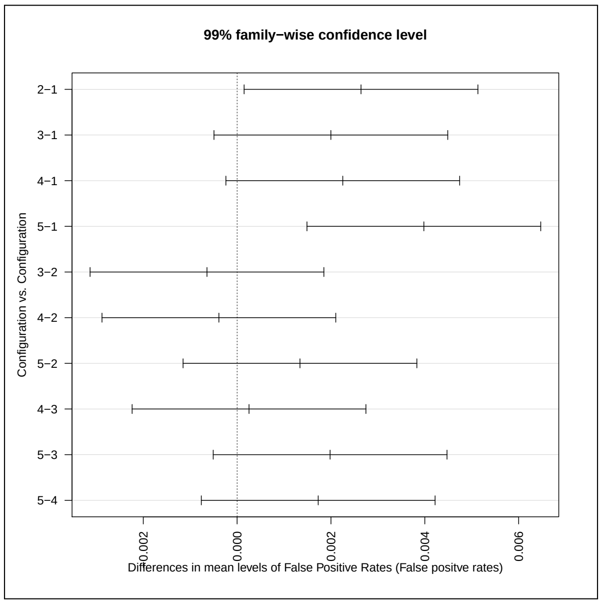


Figure B.7: Tukey HSD plot comparing different i-gram sizes by false positive rate

B.2 TPR and FPR Plots for Polymorphic and Packed Classes Combined

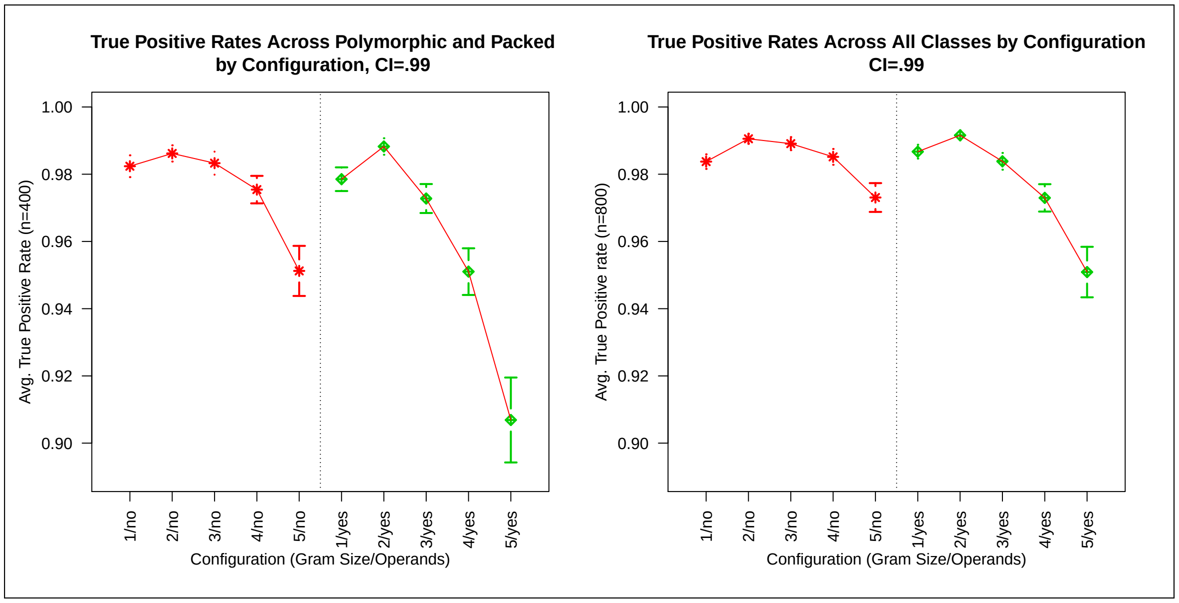


Figure B.8: Comparison of true positive rates for the polymorphic and packed classes

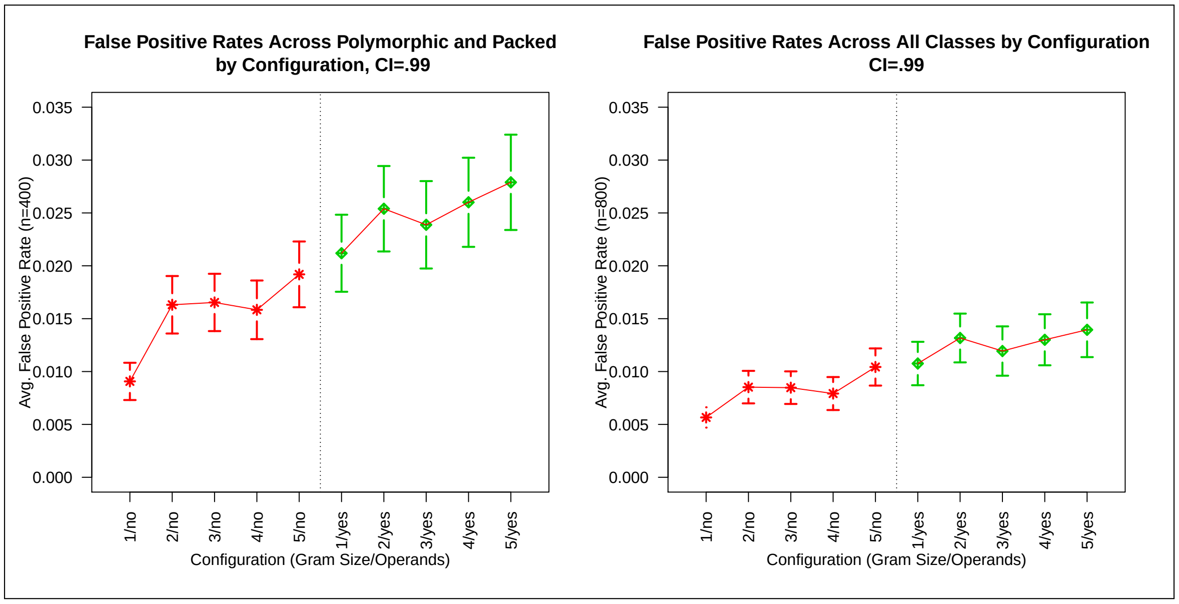


Figure B.9: Comparison of false positive rates for the polymorphic and packed classes

B.3 Accuracy, TPR, and FPR Plots for All Disassembly-based Feature Configurations

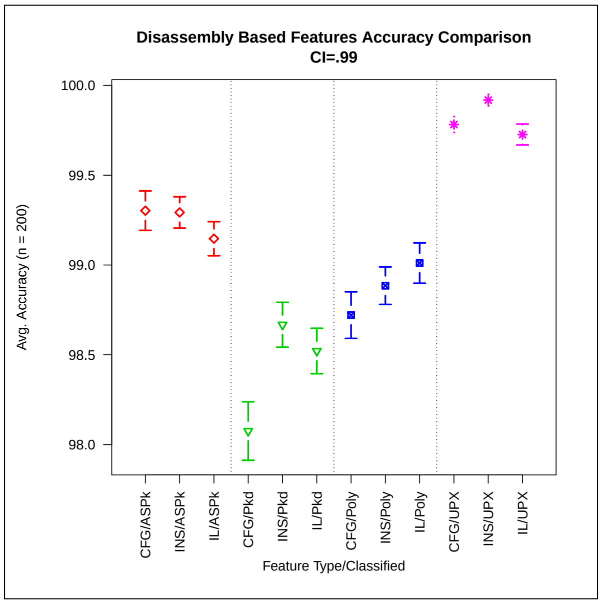


Figure B.10: Comparison of accuracies for disassembly based features

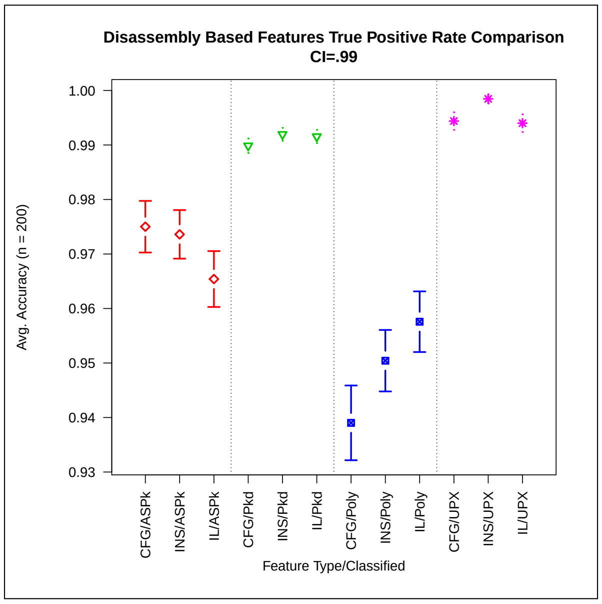


Figure B.11: Comparison of true positive rates for disassembly based features

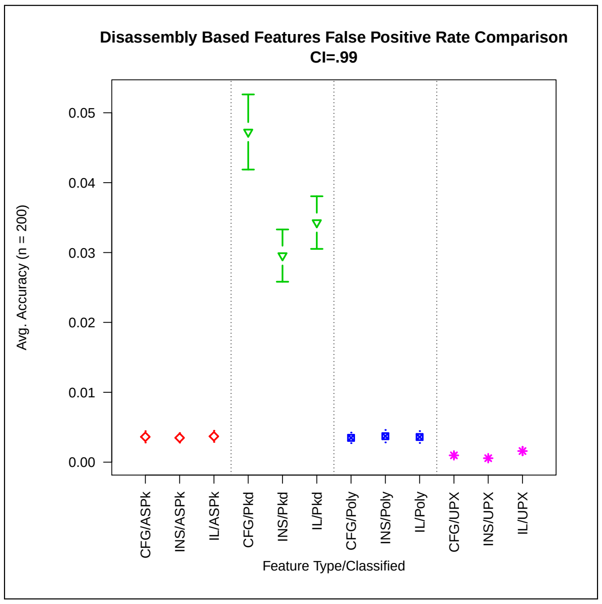


Figure B.12: Comparison of false positive rates for disassembly based features

B.4 Mean Comparisons for UPX Class Disassembly-based Feature Configurations

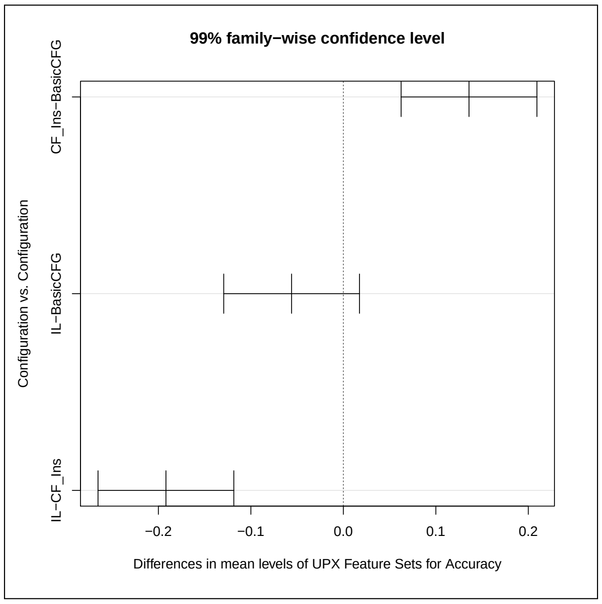


Figure B.13: Tukey HSD comparison plot of UPX accuracies for disassembly based features

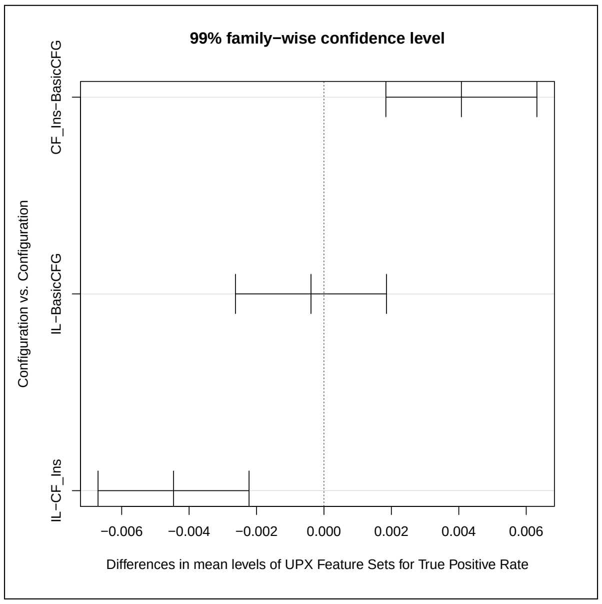


Figure B.14: Tukey HSD comparison plot of UPX true positive rates for disassembly based features

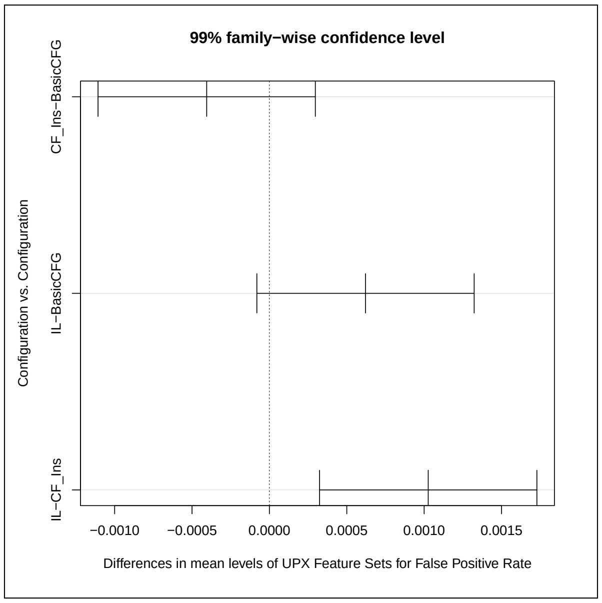


Figure B.15: Tukey HSD comparison plot of UPX false positive rates for disassembly based features

B.5 Mean Comparisons for ASPack Class Disassembly-based Feature Configurations

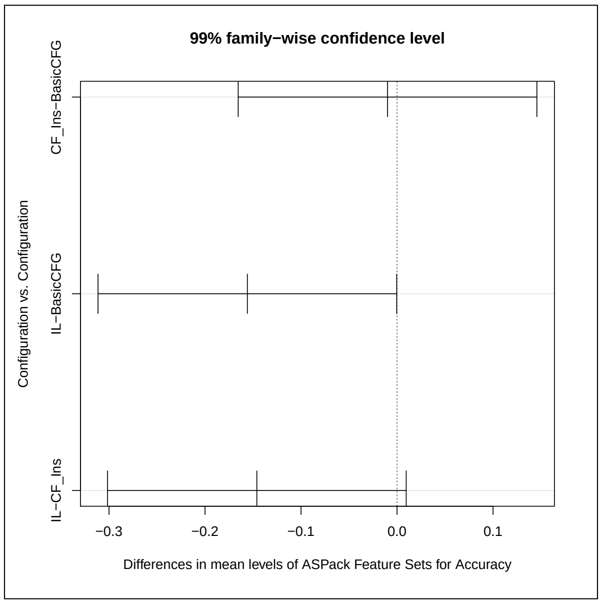


Figure B.16: Tukey HSD comparison plot of ASPack accuracies for disassembly based features

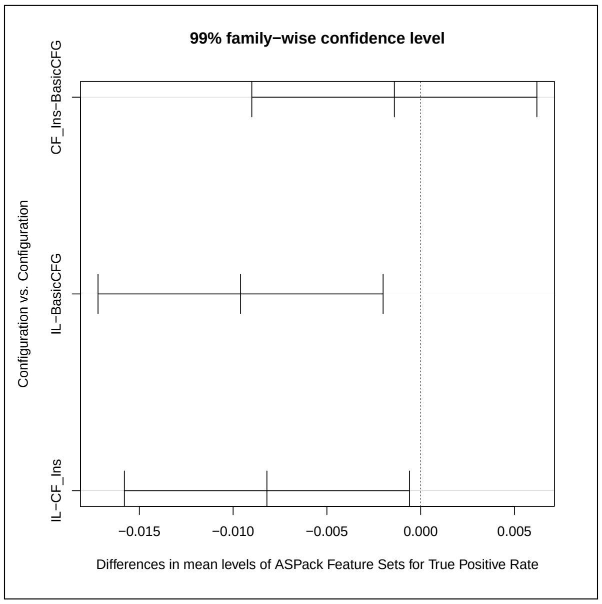


Figure B.17: Tukey HSD comparison plot of ASPack true positive rates for disassembly based features

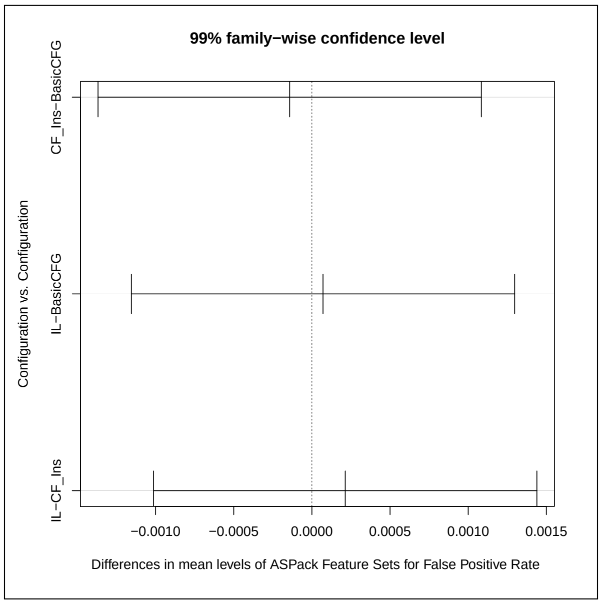


Figure B.18: Tukey HSD comparison plot of ASPack false positive rates for disassembly based features

B.6 Mean Comparisons for Polymorphic Class Disassembly-based Feature Configurations

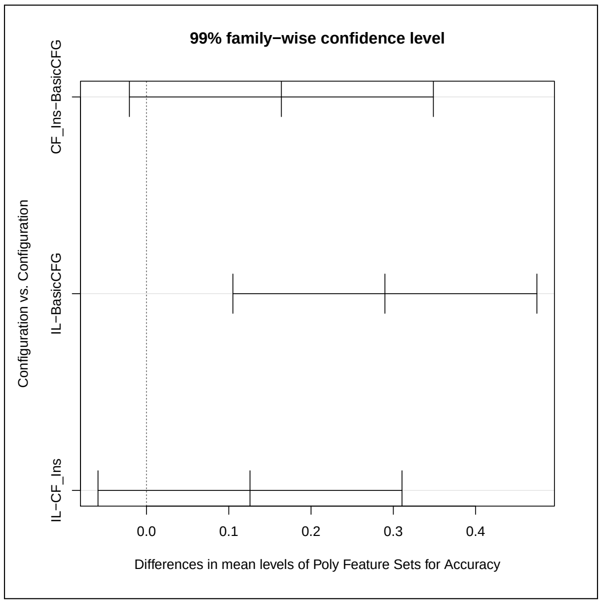


Figure B.19: Tukey HSD comparison plot of polymorphic accuracies for disassembly based features

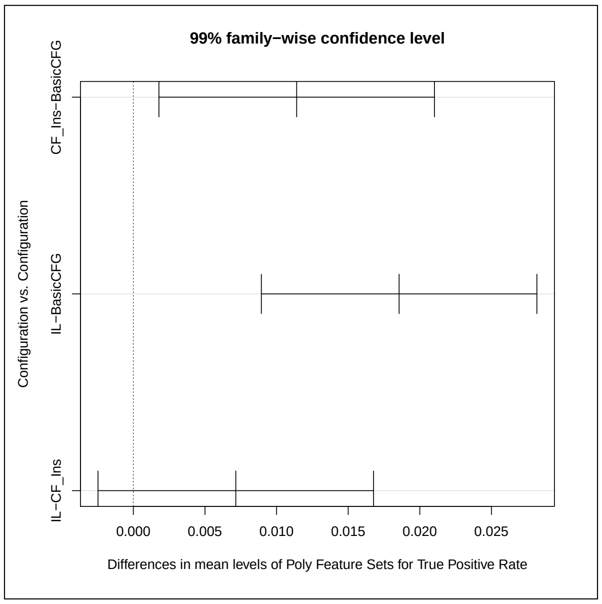


Figure B.20: Tukey HSD comparison plot of polymorphic true positive rates for disassembly based features

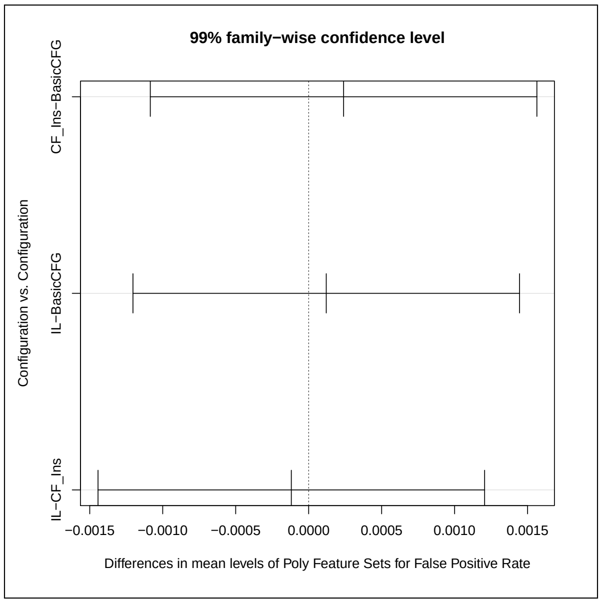


Figure B.21: Tukey HSD comparison plot of polymorphic false positive rates for disassembly based features

B.7 Mean Comparisons for Polymorphic Class Disassembly-based Feature Configurations

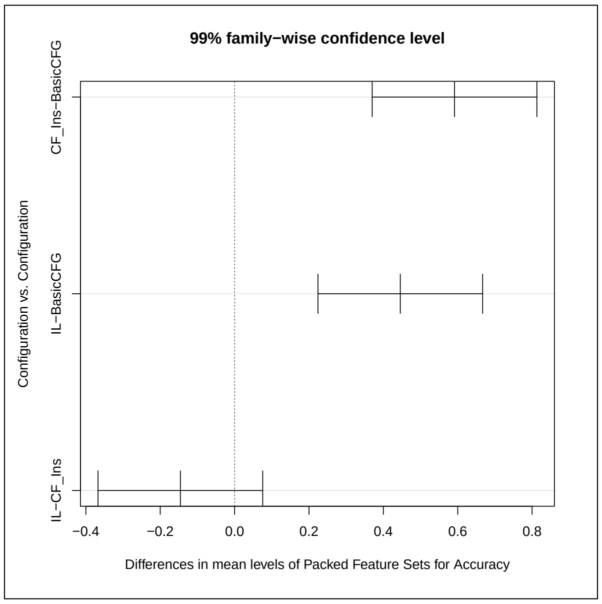


Figure B.22: Tukey HSD comparison plot of packed accuracies for disassembly based features

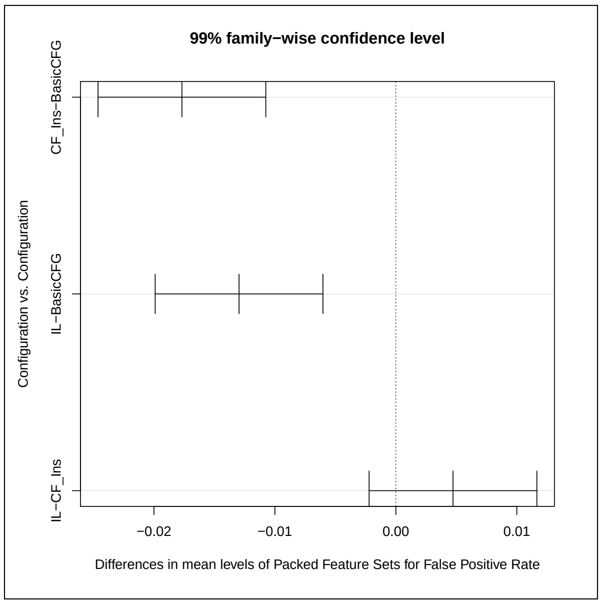


Figure B.23: Tukey HSD comparison plot of packed false positive rates for disassembly based features

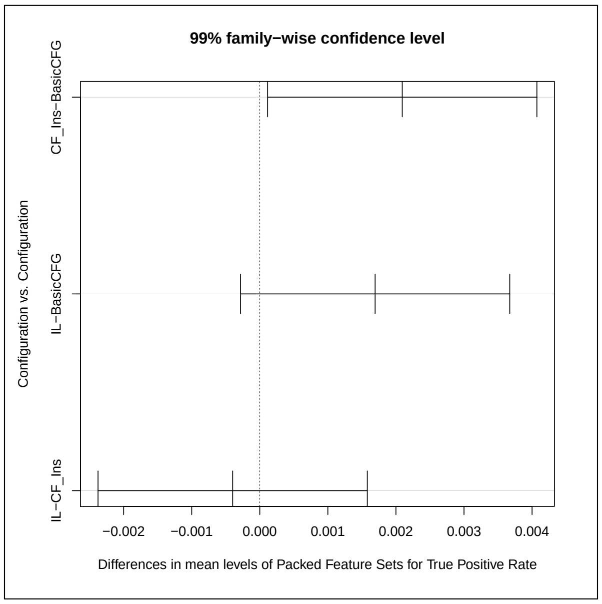


Figure B.24: Tukey HSD comparison plot of packed true positive rates for disassembly based features

APPENDIX C: Polymorphic Pilot Study Test Results for Polymorphic (msfencode) Class versus Non-Packed

This section provides the results of a pilot study conducted with all 405 of the msfencode packed executables classified against 619 of the 623 non-packed executables used in the final experiments. These results show similar performances for various i-gram configurations compared to final results gathered.

Table C.1: Average accuracies, TPRs, and FPRs with confidence intervals for i-grams on poly pilot test

Configuration	Avg. ACC	99% CI	Avg. TPR	99% CI	Avg. FPR	99% CI
POLY/1 Gram/Excluded	99.77546	99.69161 – 99.85932	0.99926	0.99816 – 1.00036	0.00323	0.002 – 0.00446
POLY/1 Gram/Included	98.79388	98.62207 – 98.96569	0.98060	0.97647 – 0.98473	0.00727	0.00485 – 0.00968
POLY/2 Gram/Excluded	99.34100	99.17223 – 99.50976	0.99840	0.9965 – 1.00029	0.00985	0.00737 – 0.01234
POLY/2 Gram/Included	98.92561	98.75459 – 99.09663	0.97777	0.97387 – 0.98167	0.00323	0.00193 – 0.00453
POLY/3 Gram/Excluded	99.24310	99.08369 – 99.40251	0.99209	0.9894 – 0.99478	0.00735	0.00531 – 0.00939
POLY/3 Gram/Included	98.36370	98.15111 – 98.57629	0.95859	0.95319 – 0.96398	0.00000	0.00000 – 0.00000
POLY/4 Gram/Excluded	97.10504	96.83368 – 97.37639	0.97087	0.96569 – 0.97606	0.02884	0.02468 – 0.033
POLY/4 Gram/Included	98.01309	97.764 – 98.26218	0.95014	0.94391 – 0.95637	0.00024	-0.00012 – 6e-04
POLY/5 Gram/Excluded	97.80273	97.54015 – 98.06531	0.99360	0.99096 – 0.99625	0.03215	0.02807 – 0.03624
POLY/5 Gram/Included	96.05054	95.70922 – 96.39187	0.98098	0.97682 – 0.98515	0.05292	0.04787 – 0.05796

Table C.2: Confusion matrix for all i-gram configurations in poly pilot test based on 10-fold, cross-validation of msfencode executables versus non-packed executables)

Configuration	TP	FP	TN	FN
POLY/1 Gram/Excluded	40.5	0.2	61.7	0.0
POLY/1 Gram/Included	39.7	0.4	61.5	0.8
POLY/2 Gram/Excluded	40.4	0.6	61.3	0.1
POLY/2 Gram/Included	39.6	0.2	61.7	0.9
POLY/3 Gram/Excluded	40.2	0.5	61.4	0.3
POLY/3 Gram/Included	38.8	0.0	61.9	1.7
POLY/4 Gram/Excluded	39.3	1.8	60.1	1.2
POLY/4 Gram/Included	38.5	0.0	61.9	2.0
POLY/5 Gram/Excluded	40.2	2.0	59.9	0.3
POLY/5 Gram/Included	39.7	3.3	58.6	0.8

BIBLIOGRAPHY

[1] *Department of Defense Strategy for Operating in Cyberspace*. DIANE Publishing, 2011.

[2] M. Christodorescu and S. Jha, "Testing malware detectors," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*. Association for Computing Machinery, 2004, pp. 34–44.

[3] M. Christodorescu, S. Jha, S. Seshia, D. Song, and R. Bryant, "Semantics-aware malware detection," in *Security and Privacy, 2005 IEEE Symposium on*, May 2005, pp. 32 – 46.

[4] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. USENIX Association, 2003, pp. 169–186.

[5] I. S. Dhillon and D. S. Modha, "Concept decompositions for large sparse text data using clustering," *Machine Learning*, vol. 42, pp. 143–175, 2001.

[6] T. Dube, R. Raines, G. Peterson, K. Bauer, M. Grimaila, and S. Rogers, "Malware target recognition via static heuristics," *Computers & Security*, vol. 31, no. 1, pp. 137 – 147, 2012.

[7] T. Dullien and S. Porst, "Reil : A platform-independent intermediate representation of disassembled code for static code analysis," *In Proceeding of CanSecWest*, 2009. [Online]. Available: http://blog.zynamics.com/2010/08/24/the-reil-language-part-iv/

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

[9] F. Guo, P. Ferrie, and T.-c. Chiueh, "A study of the packer problem and its solutions," in *Recent Advances in Intrusion Detection*. Springer Berlin / Heidelberg, 2008, vol. 5230, pp. 98–115.

[10] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: An update," *SIGKDD Explorations*, vol. 11, 2009.

[11] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, 2006.

[12] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *Recent Advances in Intrusion Detection*. Springer Berlin / Heidelberg, 2006, vol. 3858, pp. 207–226.

[13] B. D. McKay, “Practical graph isomorphism,” in *10th Manitoba Converence on Numerical Mathematics and Computing*. Congressus Numerantium, 1981, pp. 45–87.

[14] R. Moskovitch, C. Feher, N. Tzachar, E. Berger, M. Gitelman, S. Dolev, and Y. Elovici, “Unknown malcode detection using opcode representation,” in *Intelligence and Security Informatics*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, vol. 5376, pp. 204–215.

[15] G. Newsroom. (2008, June) Gartner says more than 1 billion pcs in use worldwide and headed to 2 billion units by 2014. [Online]. Available: http://www.gartner.com/it/page.jsp?id=703807

[16] M. Pietrek, “Peering inside the pe: A tour of the win32 portable executable file format,” 1994. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/ms809762.aspx

[17] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, “Polyunpack: Automating the hidden-code extraction of unpack-executing malware,” in *Computer Security Applications Conference, 2006. ACSAC ’06. 22nd Annual*, 2006, pp. 289 –300.

[18] M. Schultz, E. Eskin, F. Zadok, and S. Stolfo, “Data mining methods for detection of new malicious executables,” in *Security and Privacy, 2001. S P 2001. Proceedings. 2001 IEEE Symposium on*, 2001, pp. 38 –49.

[19] Securelist. (2012, April) Monthly malware statistics: February 2012. [Online]. Available: http://www.securelist.com/en/analysis/204792223/Monthly_Malware_Statistics_February_2012

[20] Y. seo Choi, I. kyun Kim, J. tae Oh, and J. cheol Ryou, “Pe file header analysis-based packed pe file detection technique (phad),” in *Computer Science and its Applications, 2008. CSA ’08. International Symposium on*, 2008, pp. 28 –31.

[21] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, “Detecting unknown malicious code by applying classification techniques on opcode patterns,” *Security Informatics*, vol. 1, 2012.

[22] Shadowserver.org. (2012, February) Packer statistics. [Online]. Available: http://www.shadowserver.org/wiki/pmwiki.php/Stats/PackerStatistics

[23] A. Strehl, E. Strehl, and J. Ghosh, “Value-based customer grouping from large retail data-sets,” in *In Proceedings of the SPIE Conference on Data Mining and Knowledge Discovery*, 2000, pp. 33–42.

[24] A. Sung, J. Xu, P. Chavez, and S. Mukkamala, “Static analyzer of vicious executables (save),” in *Computer Security Applications Conference, 2004. 20th Annual*, 2004, pp. 326 – 334.

[25] P. Szor, *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005.

[26] J.-Y. Xu, A. Sung, P. Chavez, and S. Mukkamala, “Polymorphic malicious executable scanner by api sequence analysis,” in *Hybrid Intelligent Systems, 2004. HIS ’04. Fourth International Conference on*, December 2004, pp. 378 – 383.

[27] W. Yan, Z. Zhang, and N. Ansari, “Revealing packed malware,” *Security Privacy, IEEE*, vol. 6, no. 5, pp. 65 –69, sept.-oct. 2008.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE *(DD-MM-YYYY)* 14-06-2012	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From - To)* Sep 2010 - Jun 2012

4. TITLE AND SUBTITLE Intra-procedural Path-insensitve Grams (i-grams) and Disassembly Based Features for Packer Tool Classification and Detection	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Gerics, Scott E., Captain	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, Graduate School of Engineering and Management (AFIT/EN), 2950 Hobson Way, Wright-Patterson AFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT/GCE/ENG/12-07
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) INTENTIONALLY LEFT BLANK	**10. SPONSOR/MONITOR'S ACRONYM(S)**
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The DoD relies on over seven million computing devices worldwide to accomplish a wide range of goals and missions. Malicious software, or malware, jeopardizes these goals and missions. However, determining whether an arbitrary software executable is malicious can be difficult. Obfuscation tools, called packers, are often used to hide the malicious intent of malware from anti-virus programs. Therefore detecting whether or not an arbitrary executable file is packed is a critical step in software security. This research uses machine learning methods to build a system, the Polymorphic and Non-Polymorphic Packer Detection (PNPD) system, that detects whether an executable is packed using both sequences of instructions, called i-grams, and disassembly information as features for machine learning. Both i-grams and disassembly features successfully detect packed executables with top configurations achieving average accuracies above 99.5\%, average true positive rates above 0.977, and average false positive rates below 1.6e-3 when detecting polymorphic packers.

15. SUBJECT TERMS

^Machine learning, n-grams, static analysis, executable packers, packer detection, software obfuscation

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Rusty O. Baldwin, ENG
U	U	U	UU	98	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-3636 x4445; Rusty.Baldwin@afit.edu

Reset