Abstract

Malware is a pervasive problem in distributed computer and network systems. Identification of malware variants provides great benefit in early detection. Control flow has been proposed as a characteristic that can be identified across variants, resulting in classification employing flowgraph based signatures. Static analysis is widely used to construct the signatures but can be ineffective if malware undergoes a code packing transformation to hide its real content. This thesis proposes a novel system, named Malwise, for malware classification using a fast application level emulator to reverse the code packing transformation, and two flowgraph matching algorithms to perform classification: exact flowgraph matching and approximate flowgraph matching. The exact flowgraph matching algorithm uses string based signatures of graph invariants, and is able to detect malware with near real-time performance. The approximate flowgraph matching algorithm is slower but more effective and uses the decompilation technique of structuring to generate string based signatures amenable to comparisons using the string edit distance. To demonstrate the effectiveness and efficiency of the automated unpacking and flowgraph based classification, we evaluate the system with synthetic malware and over 15,000 real samples. The evaluation shows our system is highly effective in terms of accuracy in revealing all a sample's hidden code, execution time for unpacking and classification, and accuracy in detection of malware variants.

Fast Automated Unpacking and Classification of Malware

Silvio CESARE

Master of Informatics

School of Management and Information Systems

Faculty of Arts, Business, Informatics and Education

Central Queensland University

May 2010

Certificate of Authorship and Originality of thesis

The work contained in this thesis has not been previously submitted either in whole or in part for a degree at Central Queensland University or any other tertiary institution. To the best of my knowledge and belief, the material presented in this thesis is original except where due reference is made in text.

Signed:

Date: 17 May 2010

Copyright statement

Signed:

Date: 17 May 2010

:

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would first like to thank my family for their support and in particular my mother, Maxine Cesare, whose assistance enabled me to have the opportunity to pursue my ambitions of academic study. I am also thankful for the support of my principal supervisor, Dr Yang Xiang, who provided continual assistance, including administrative and scholarship support. I am thankful for his desire to publish during candidature.

List of Publications

[1] Silvio Cesare, and Yang Xiang 2010, *Classification of Malware Using Structured Control Flow*, 8th Australasian Symposium on Parallel and Distributed Computing (AusPDC 2010).

[2] Silvio Cesare, and Yang Xiang 2010, *A Fast Flowgraph Based Classification System for Packed and Polymorphic Malware on the Endhost*, IEEE 24th International Conference on Advanced Information Networking and Application (AINA 2010), IEEE.

1 Introduction

1.1 Background to the Study

The presence of malicious software is problem that plagues internet and network connectivity. Malicious software, also known as malware, are programs characterised by their malicious intent. They are hostile, intrusive or annoying software programs. Examples of malware include trojan horses, worms, backdoors, dialers and spyware. Malware is a problem that is increasing at a significant rate. According to the Symantec Internet Threat Report [1], 499,811 new malware samples were received in the second half of 2007. F-Secure additionally reported, “As much malware [was] produced in 2007 as in the previous 20 years altogether“ [2].

The modern purpose of malware is that of criminal enterprise for financial gain [3]. In 2008, “78 percent of confidential information threats exported user data” [3]. The stealing of banking information using malware known as spyware [4] to covertly log and relay such private information, is a common example of modern malware.

The malware problem continues when malicious software remains undetected by users. The creation of criminal networks employing unauthorised use of users’ computers is an example of a malicious botnet [5]. Botnets are illegally leased to criminal networks in order to create Email spamming networks, and to extort money from commercial entities using the threat of distributed denial of service attacks. A user’s inability to prevent or detect malware often makes them liable to become an additional node in a botnet’s zombie network.

Detection of malicious software provides much benefit in fighting the threat that malware poses to users' security. Detecting malware before it is allowed to execute its intent allows such software to be effectively disabled. To identify a program as being malicious or benign, automated analysis is required. The analysis can employ either a static or dynamic approach. In the dynamic approach, the malware is executed, possibly in a sandbox, and its runtime behaviour examined. In the purely static approach, the malware is never executed.

Traditional Antivirus solutions to secure systems against malware have focused on static detection. Dynamic approaches [6], while having some benefits compared to static detection, also have disadvantages. The dynamic approach requires an execution environment in which to run, mandating that a virtual machine or sandbox is available. For cross platform systems, this may be an ineffective environment in which to operate. If a virtual environment is not provided, execution of the malware on the host is required, which may allow malware to execute its intent, before being detected. Additionally, a dynamic analysis may fail to identify malicious software if the malicious behaviour is not triggered during the analysis. While dynamic malware detection is an important topic, this thesis focuses only on the static detection of malware.

Traditional solutions to static malware detection have employed the use of signatures. Signatures capture invariant characteristics, or fingerprints, in the malware that uniquely identify it. Because of performance constraints, the most predominantly used signature is a string containing patterns of the raw file content [7, 8]. This allows for a string search [9] to quickly identify patterns associated with known malware. However, these patterns can easily be invalidated because minor changes to the malware source code have significant effects on the malware's raw content. Thus, traditional signatures can prove ineffective when dealing with unknown variants.

Malware authors attempt to evade detection by creating polymorphic variants of their malware which are not detected by anti-malware systems. Polymorphism describes related, but different instances of malware sharing a common history of code. Code sharing among variants can have many sources, whether derived from autonomously self mutating malware, or manually copied by the malware creator to reuse previously authored code. Related to polymorphic malware are packed malware. Code packing is an obfuscation technique used to hide a malware's real content. A code packing tool is applied to a malware instance, as a post-processing binary rewriting stage, to produce a new packed version of the malware. It is often used to make manual analysis and automated analysis of the malware more difficult. Code packing is also used to evade signature detection by Antivirus software through the creation of malware variants which have no associated Antivirus signature.

For a malware detection system to perform effectively, packed and polymorphic malware variants must be detected. The detection of polymorphic malware has generated an interest in classifying malware using features at a higher level abstraction than the traditional byte level content. The field of static program analysis has provided benefits to malware detection.

This thesis investigates unpacking combined with the static detection and classification of malware and their polymorphic variants. We systematically examine packing and polymorphism, develop algorithms to unpack malware, and develop algorithms to classify malicious software derived from known malware.

1.2 Aim and Scope of the Research

1.2.1 Aim

The aim of this research is to discover effective and efficient methods for the detection and classification of malware.

1.2.2 Scope

The scope of this study is limited to malware in the form of executable program instances. Network intrusion detection of malicious content, or the general detection of worms based on network traffic, while important in their own right, are not investigated. To achieve the aim of detecting and classifying malware, the associated analyses must incorporate the removal of obfuscations incorporated by the malware creator(s) that would otherwise make such analyses ineffective. This mandates that the code packing obfuscation in malware, and the unpacking of such obfuscated malware, be investigated. The scope is limited to only the code packing transformations and obfuscations evident in malware. The scope does not extend into the general problem of deobfuscation and associated static analyses. The malware detection and classification, subsequent to the code unpacking, is limited to only static analyses.

1.3 Structure of the Thesis

This thesis is organized as follows.

- Chapter 2 surveys the related work of malware unpacking and static classification of malware. Insight is provided into how we chose to approach the problem of malware classification. The survey of related work specifically identifies the potential of application level emulation in contrast to the existing approaches in automated unpacking. The survey also examines the benefits of control flow as a static malware feature.
- Chapter 3 precisely defines the malware classification problem we aim to address. We specifically examine the problem of malware variant detection based on constructing similarity between programs. This chapter also outlines the general approach for Malwise, our prototype malware classification system.
- Chapter 4 provides our proposed automated unpacking system based on application level emulation. We also propose our method for detecting when unpacking is complete using the technique of entropy analysis. We evaluate the effectiveness and efficiency of the prototype unpacking system used in Malwise.
- Chapter 5 examines the static features we extract from malware that we will use to classify malware. We propose two novel feature sets based on representing control flow graphs as strings.
- Chapter 6 proposes our classification and database search algorithm based on identifying similarity between programs. We perform an evaluation of the effectiveness and efficiency of our prototype system, Malwise, that performs unpacking and malware classification.
- Chapter 7 summarizes and concludes the thesis.

1.4 Major Contributions of the Thesis

The major contributions of this thesis are as follows:

- We propose, implement and evaluate the use of application level emulation in automated unpacking. Application level emulation has commercial interest, but has previously lacked academic evaluation.
- We propose the use of entropy analysis to detect that automated unpacking is complete and the hidden code has been revealed. Entropy analysis has previously been used to detect packed binaries, but has not been used in malware unpacking.
- We propose using a graph invariant based signature to estimate control flow graph isomorphism for the purpose of constructing a measure of program similarity. The graph invariant chosen has been used previously to aid detection of malware, but has not been used as a dominant feature in malware classification.
- We propose using the decompilation technique of structuring to generate a string based control flow signature, amenable to comparisons using the string edit distance. This approach can be used for approximate control flow graph matching. Decompilation has not been used previously to construct control flow graph signatures or to perform malware classification
- We propose a set similarity function and a set similarity search algorithm which form the basis for our malware classification system and which perform

efficiently in the expected case. The set similarity function and search are unique to our work.

- We implement and evaluate our ideas in a novel prototype system named Malwise.

2 Related Work

This chapter surveys the related work in malware unpacking and classification. The structure of this chapter is as follows:

- Taxonomy of the techniques used to create polymorphic malware variants is described in Section 2.1.
- Section 2.2 examines the code packing transformation technique used in polymorphic malware in more detail. The code packing transformation is used primarily as an obfuscation technique for malware.
- Section 2.3 provides taxonomy of static features that are present in malware and benign samples that can be used for automated malware classification and detection purposes.
- Section 2.4 compares the program features identified in the taxonomy.
- Section 2.5 categorizes the taxonomy of static program features in terms of their abstract models and mathematical representations.
- Section 2.6 examines static analysis techniques that can be used on malware and benign programs, such as disassembly and control flow reconstruction.
- The automated unpacking of samples is examined in Section 2.7.
- Section 2.8 then surveys the literature that investigates static classification of malware when control flow is used as a feature.
- Section 2.9 examines future trends.

- Section 2.10 summarizes the chapter.

2.1 Taxonomy of Malware Polymorphism

2.1.1 Syntactic Changes

A syntactic polymorphic malware technique is a method that changes the syntactic structure of the malware [10]. Though the syntactic structure changes in polymorphic malware, the malware semantically remains identical. The technique is predominantly used to evade byte level signature based detection and classification that is routinely employed by traditional Antivirus. Polymorphism borrows many of the techniques from the field of program obfuscation.

Polymorphism is sometimes described by the similar term of metamorphism. In that usage it is used to describe the automated syntactic mutation of the malware's code and instructions. Under such terminology, polymorphism is used to describe syntactic mutation of limited parts of the malware's instruction content. The remaining parts of the malware are encoded at the byte level without regard to the instruction syntax or semantics. In this survey we treat polymorphism and metamorphism as identical to each other.

2.1.1.1 Dead Code Insertion

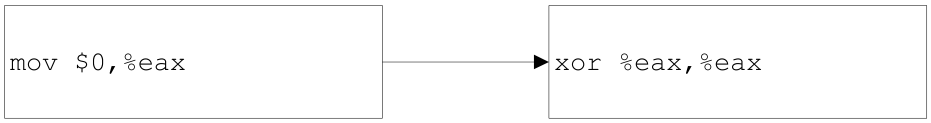
Dead code is also known as junk code and a semantic nop [10]. Dead code is semantically equivalent to a nil operation. Insertion of this type of code has no semantic impact on the malware. The insertion increases the size of the malware and modifies the byte and instruction level content of the malware.

```
push %ebx
pop %ebx
```

Figure 1. A semantic nop.

2.1.1.2 Instruction Substitution

Instruction substitution replaces specific instructions or sequences of instructions with semantically equivalent, but differing instructions and instruction sequences. The size of the malware may grow or shrink in this procedure.



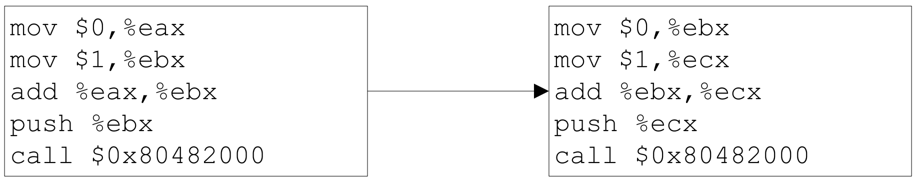
```
mov $0,%eax
```

```
xor %eax,%eax
```

Figure 2. Instruction substituion.

2.1.1.3 Variable Renaming

Variable renaming [11] and the associated technique of register reassignment alters the use of variables and registers in a sequence of code such that the instructions are semantically equivalent but use different variables and registers when compared to the original code.



```
mov $0,%eax
mov $1,%ebx
add %eax,%ebx
push %ebx
call $0x80482000
```

```
mov $0,%ebx
mov $1,%ecx
add %ebx,%ecx
push %ecx
call $0x80482000
```

Figure 3. Register reassignment.

2.1.1.4 Code Reordering

Code reordering [11] changes the syntactic order of the code in the malware [10]. The actual or semantic execution path of the program does not change. However, the syntactic order as present in the malware image is altered. Code reordering includes the techniques of branch obfuscation, branch inversion, branch flipping, and the use of opaque predicates.

2.1.1.5 Branch Obfuscation

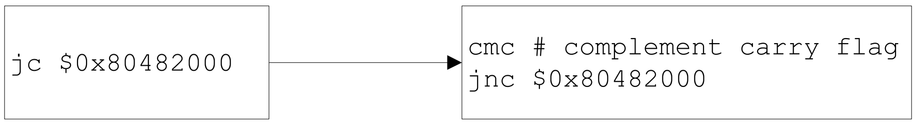
Branch obfuscation attempts to hide the target of a branch instruction. Examples include the use of Structured Exception Handling (SEH) on the Microsoft Windows platform. The use of SEH to obscure control flow is common in modern malware. Similar techniques involve indirect branching. Indirect branching uses data content as the target of a branch. This translates control flow identification into a harder data flow analysis problem. The use of a branch function [12] extends this approach and dispatches multiple branches through a single routine. The main purpose of branch obfuscation is to make the static analysis of the malware by an analyst or automated system harder to perform.

```
mov $0x8048200,%eax
jmp *%eax
```

Figure 4. An indirect branch.

2.1.1.6 Branch Inversion and Flipping

Branch inversion inverts the branch condition in conditional branches. Whereas the branch may originally transfer control when the condition is true, branch inversion alters the condition to branch when false. To maintain the original semantics of the program the branch instruction is also inverted. For example, a branch on condition true statement would be changed to a branch on condition false statement. Additionally, the condition being tested would also be inverted. Branch inversion is effectively a form of instruction substitution on control flow statements.

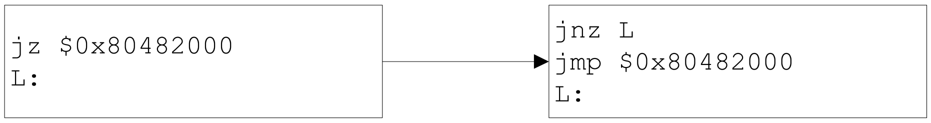


```
jc $0x80482000
```

```
cmc # complement carry flag
jnc $0x80482000
```

Figure 5. Branch inversion.

Branch flipping [12] is a similar technique to branch inversion and rewrites the branch instruction by substituting it with semantically equivalent code with different control flow properties. For example, if the original code is to branch on condition true then the new code branches on condition false to the original fall-through instruction. The new fall-through instruction then unconditionally branches to the original conditional branch target.



```
jz $0x80482000
L:
```

```
jnz L
jmp $0x80482000
L:
```

Figure 6. Branch flipping.

2.1.1.7 Opaque Predicate Insertion

An opaque predicate [12] is a predicate that always evaluates to the same result. An opaque predicate is constructed so that it is difficult for an analyst or automated analysis to know the predicate result. Opaque predicates can be used to insert superfluous branching in the malware's control flow. They can also be used to assign variables values which are hard to determine statically. The use of opaque predicates is primarily for code obfuscation, and to prevent understanding by an analyst or automated static analysis.

```
mov $1,%eax
jz $0x80482000
```

Figure 7. A simplified opaque predicate.

2.1.1.8 Code Packing

Code packing [13, 14] is used to hide and obfuscate the contents of malware from an analyst and automated static analyses. Code packing is described in Section 2.2.

2.1.2 Semantic Changes

An extension to syntactic polymorphism is that of semantic polymorphism where the new variant is a derived work of the original malware. Semantic changes to malware occur due to the malware authors modifying the original source code or functionality. This can occur to a natural evolution of the malware during its software development life cycle. Additionally, it can occur when a malware author reuses existing malicious code in a new malware instance.

2.1.2.1 Code Insertion

Code insertion occurs when new functionality is added to the malware.

2.1.2.2 Code Deletion

Code deletion occurs when functionality is removed from the malware.

2.1.2.3 Code Substitution

Code substitution occurs when functionality in the malware is replaced by an alternative algorithm or code.

2.1.2.4 Code Transposition

Code transposition occurs when specific code and functionality of the malware is removed from its initial location and inserted into a semantically different location in the malware.

2.2 Malware Obfuscation Using Code Packing

Code packing is the dominant technique used to obfuscate malware and hinder an analyst's understanding of the malware's intent. In one month during 2007, 79% of identified malware from a commercial Antivirus vendor was found to be packed [15]. Additionally, almost 50% of new malware in 2006 were repacked versions of existing malware [16].

Code packing, in addition to obfuscating the understanding of the malware by an analyst, is also used by malware to evade an Antivirus system's detection. Polypack

[17] evaluated the effectiveness of code packing against Antivirus detection by providing a service to pack malware using a variety of code packing tools. Antivirus systems often have the capabilities of unpacking known code packing tools, and unpacking unknown tools has also had commercial interest [18]. However, Polypack demonstrated that packing can be an effective tool to defeat an Antivirus system with many commercial malware detection systems failing to identify the packed versions of existing malware.

Code packing is used in the majority of malware, but code packing also serves to provide compression and software protection for the intellectual property contained in a program. It is not necessarily advantageous to flag all occurrences of code packing as being indicative of malicious activity. Code packing tools are freely available [19] and commercially sold to the public as legitimate software [20]. For this reason, unpacking of packed programs provides benefit. It is advisable to determine if the packed contents are malicious, rather than identifying only the fact that unknown contents are packed.

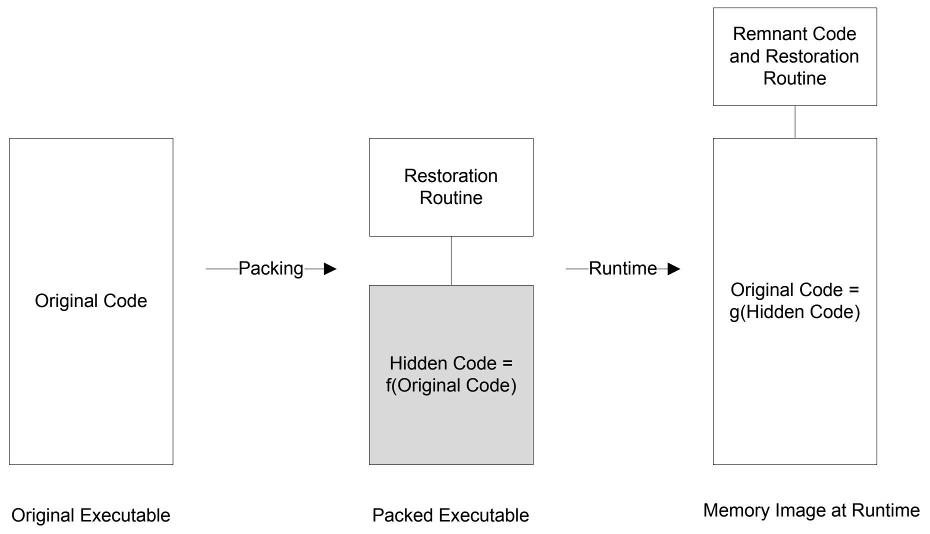


Figure 8. The traditional code packing transformation.

2.2.1 Traditional Code Packing

The most common method of code packing is described in [13]. Malware employing this method of code packing transforms executable code into data as a post-processing stage in the malware development cycle. This transformation may perform compression or encryption, hindering an analyst's understanding of the malware when using static analysis. At runtime, the data, or hidden code, is restored to its original executable form through dynamic code generation using an associated restoration routine [21]. Execution then resumes as normal to the original entry point. The original entry point marks the entry point of the original malware, before the code packing transformation is applied. Execution of the malware, once the restoration routine is complete and control is transferred to the original entry point, is transparent to the fact that code packing and restoration had been performed. A malware may have the code packing transformation applied more than once. After the restoration routine of one packing transformation has

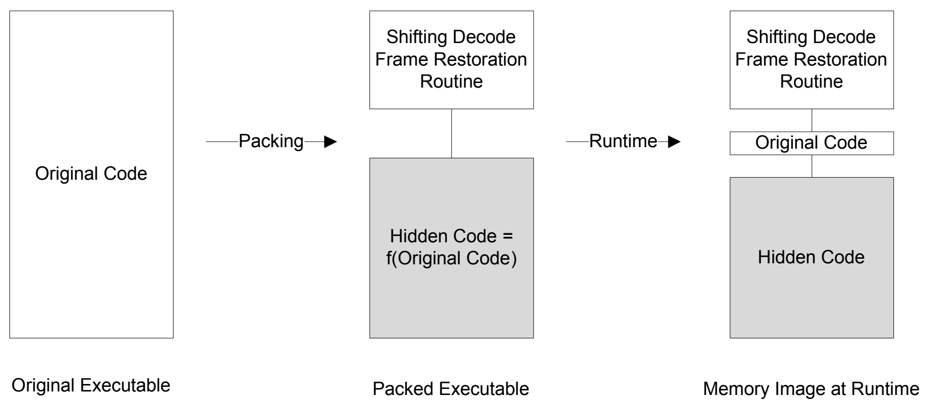

Original Code
Packing
Shifting Decode Frame Restoration Routine
Hidden Code = f(Original Code)
Runtime
Shifting Decode Frame Restoration Routine
Original Code
Hidden Code
Original Executable
Packed Executable
Memory Image at Runtime

Figure 9. Code packing using the shifting decode frame.

been applied, control may transfer another packed layer. The original entry point is derived from the last such layer.

2.2.2 Shifting Decode Frame

An extension to traditional code packing is to maintain as much of the packed image in an encrypted form at run-time. During execution of the malware, blocks of memory can be decrypted as needed and subsequently re-encrypted to prevent an analyst or automated system from having access to all the hidden code at any single moment in time. This technique is known as the shifting decode frame [22]. The granularity of encryption can occur at the page level, the basic block level, and the instruction level. This type of code packing is not often used in wild malware, and in practice, traditional code packing and instruction virtualization are the dominant techniques used in real malware.

2.2.3 Instruction Virtualization and Malware Emulators

Code packing may employ the use of instruction virtualization also known as a malware emulator [14]. An emulator used by a malware should not be confused with an emulator

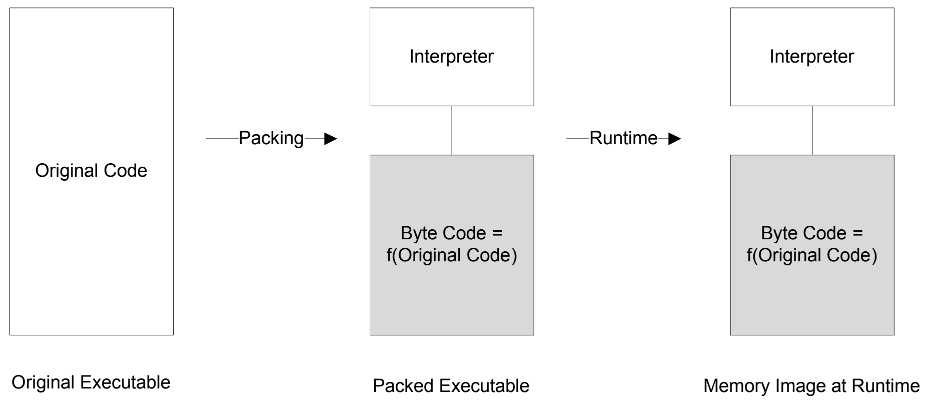


Figure 10. Code packing using instruction virtualization.

used for automated unpacking of the malware. This type of code packing transformation employing an emulator is used in a minority of malware. In this form of code packing, packing translates the original native code into a byte-code which is subsequently emulated by the malware at run-time. Using this form of code packing, the hidden code in its original form is never revealed.

2.2.4 Resistance to Dynamic Analysis

Many malware packers introduce code that intentionally makes run-time analysis of the packed malware more difficult [22]. Strategies employed by packed malware include detection of the malware being debugged, or detection of the malware being executed inside a virtual machine. These techniques are currently being employed by malware [23]. In these situations, when an attempted dynamic analysis is being performed, the execution of the malware packer diverges and the true malware behavior remains hidden without execution.

2.3 Taxonomy of Static Program Features

Malware classification and detection involves the extraction of features which are subsequently used to characterize the malware. Features may be extracted dynamically or statically. Dynamic approaches to malware classification involve monitoring execution of the programs and extracting features based on their behaviour. Static approaches extract features without program execution.

2.3.1 Object File Header Attributes

The object file header contains attributes which are often custom written during link editing and binary rewriting.

2.3.2 Bytes

The simplest feature that can be extracted from a program is the raw byte level content of the malware executable file [24]. An alternative source of content comes from the individual program sections in the binary, including the code and data segments.

```
8d 4c 24 04
83 e4 f0
ff 71 fc
55
89 e5
51
83 ec 24
e8 6a 00 00 00
c7 45 f8 00 00 00 00
eb 10
c7 04 24 a0 20 40 00
e8 5d 00 00 00
83 45 f8 01
83 7d f8 09
7e ea
83 c4 24
59
5d
8d 61 fc
c3
```

```
lea     0x4(%esp),%ecx
and     $0xfffffff0,%esp
pushl   -0x4(%ecx)
push    %ebp
mov     %esp,%ebp
push    %ecx
sub     $0x24,%esp
call    4011b0 <___main>
movl    $0x0,-0x8(%ebp)
jmp     40115f <_main+0x2f>
movl    $0x4020a0,(%esp)
call    4011b8 <_puts>
addl    $0x1,-0x8(%ebp)
cmpl    $0x9,-0x8(%ebp)
jle     40114f <_main+0x1f>
add     $0x24,%esp
pop     %ecx
pop     %ebp
lea     -0x4(%ecx),%esp
ret
```

```
lea     0x4(%esp),%ecx
and     $0xfffffff0,%esp
pushl   -0x4(%ecx)
push    %ebp
mov     %esp,%ebp
push    %ecx
sub     $0x24,%esp
call    4011b0 <___main>
movl    $0x0,-0x8(%ebp)
jmp     40115f <_main+0x2f>
```

```
movl    $0x4020a0,(%esp)
call    4011b8 <_puts>
addl    $0x1,-0x8(%ebp)
```

```
cmpl    $0x9,-0x8(%ebp)
jle     40114f <_main+0x1f>
```

```
add     $0x24,%esp
pop     %ecx
pop     %ebp
lea     -0x4(%ecx),%esp
ret
```

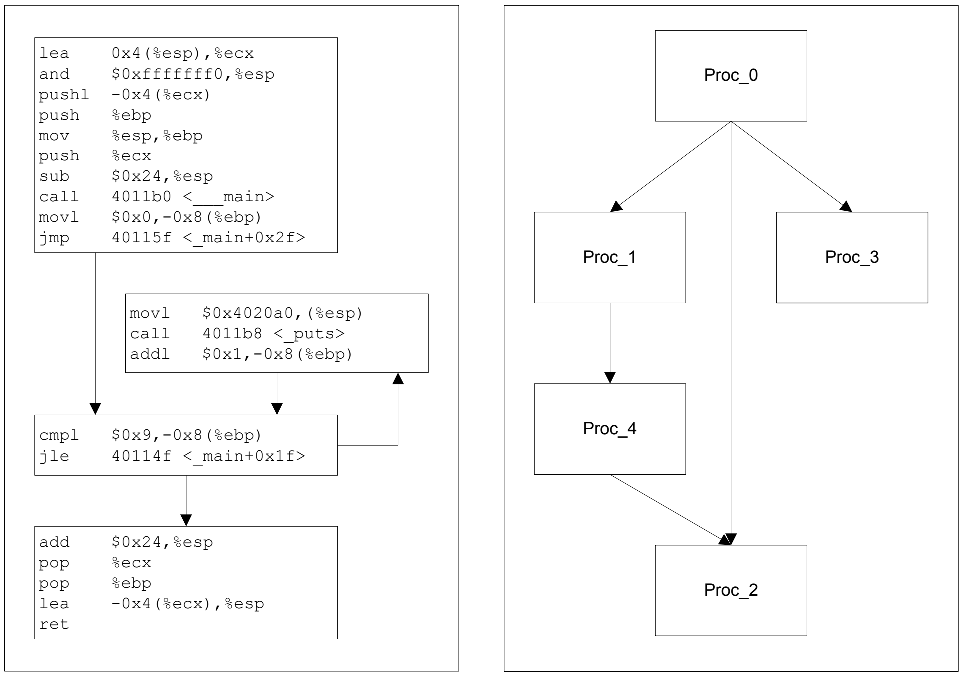
Figure 11. An example of basic blocks and instructions in a program.

2.3.3 Instructions

An executable program is constructed of code and data. The code is represented as assembly language. Extracting the assembly is the process of disassembling. The instruction level content of a program can represent a more resilient form than the byte level content if the instructions are considered by their type or mnemonic representation [25].

2.3.4 Basic Blocks

A basic block is a straight line sequence of code without an intervening control transfer instruction [26]. The basic block may be treated at the byte level, or at the instruction level. Additionally, data dependencies within the basic block may be examined to construct a directed acyclic graph [27]. The basic blocks may also be grouped to form a set, or they may have additional structure imposed by the control flow graph.



```
lea     0x4(%esp),%ecx
and     $0xfffffff0,%esp
pushl   -0x4(%ecx)
push    %ebp
mov     %esp,%ebp
push    %ecx
sub     $0x24,%esp
call    4011b0 <___main>
movl    $0x0,-0x8(%ebp)
jmp     40115f <_main+0x2f>
```

```
movl    $0x4020a0,(%esp)
call    4011b8 <_puts>
addl    $0x1,-0x8(%ebp)
```

```
cmpl    $0x9,-0x8(%ebp)
jle     40114f <_main+0x1f>
```

```
add     $0x24,%esp
pop     %ecx
pop     %ebp
lea     -0x4(%ecx),%esp
ret
```

Figure 12. A control flow graph (left), and a call graph (right).

2.3.5 Control Flow Graphs

The control flow graph is a directed graph, where the nodes are basic blocks [28]. The edges in the graph represent the possible control flow of the associated procedure. The control flow graph represents the intra-procedural control flow. A program may be considered a set of control flow graphs, or the control flow graphs may have additional structure as dictated by the call graph. Alternatively, control flow graphs may represent inter-procedural and intra-procedural control flow in a single graph. In this case, the graph represents the whole program control flow graph.

It is possible to construct alternative or abstracted representations of the control flow graph. Loop nest trees, dominator trees, and control dependency graphs can also be constructed [27].

2.3.6 Call Graph

Call graphs like control flow graph model the possible execution paths and control flow in a program [29]. The call graph is a directed graph representing the inter-procedural control flow.

Like the control flow graph, alternative or abstracted representations are possible such as a dominator tree.

2.3.7 API Calls

Programs interface with the underlying operating system and libraries. The invocation of an API function from a known library can often be identified statically [30]. The API call sequence gives insight to the behaviour of the program.

2.3.8 Data Flow

The data flow of a program represents the set of possible values data may hold during program execution [31]. Many types of data flow analyses exist, including live variable analysis, reaching definitions, and value-set analysis. Each analysis looks at a particular property of the data at specific program points. Modelling the data flow requires that the control flow be successfully identified. A simpler model of data dependencies can be modelled as described in the basic block feature section.

2.3.9 Procedure Dependence Graph

A procedure dependency graph combines the control dependencies and data dependencies of a procedure into a single graph.

2.3.10 System Dependence Graph

The system dependence graph is a collection of procedure dependence graphs; one for each procedure in the program.

2.4 Comparison of Static Program Features

Malware may be polymorphic, but static program features are known to be invariant under different polymorphic techniques.

Byte and instruction level program features perform poorly when faced with the polymorphic variations and mutations. Recompiling source code using different compile time options may result in syntactic changes including variable renaming, and instruction substitution. Code normalization [10] can sometimes reverse the effects of syntactic polymorphism and can work in practice, but is not based on a sound technique. Additionally, the byte and instruction stream may change when minor semantic alterations are made to the malware source code.

The advantage of byte level content as a program feature is that the dependence on accurate static analysis of the programs semantics or structure is not required.

If the instruction stream is used, additional challenges are presented because it is known that perfect disassembly of an unknown image is undecidable on the x86 platform [32].

To avoid the problems of syntactic polymorphism, higher level abstractions of the program can be used. The control flow features including control flow graphs and call graphs are considered more invariant in polymorphic malware than byte and instruction level content [28]. However, opaque predicates may result in these features being altered. The detection of opaque predicates has been investigated, but it is not evident that this is entirely satisfactory, and a sound method of detection against all unknown predicates is not possible. For example, it is known that some algorithms which are used to construct predicates are not proven to be true and remain only as conjectures that produce the same predicate under current testing.

The presence of pointers and indirection in assembly language also present problems to static analyses which may not have the precision required to construct a control flow graph or call graph with the degree of accuracy required for malware classification. For all its disadvantages, control flow has shown to be an effective feature that is invariant in most current malware.

The use of API calls is another approach to solve the syntactic polymorphism problem. This approach has problems with malware that obscures the use of those calls, as is the case of the stolen bytes technique [22] introduced by code packing tools.

Data flow analysis is another high level abstraction but when used in the presence of pointers is compounded by the problems that static analyses must face.

The procedure and system dependence graphs have similar problems with pointers and indirection even when the data dependencies of pointers are ignored. The dependence graphs are also dependent on accurate modelling of the instruction sequence. This avoids problems such as register reassignment because the data dependencies are represented as a graph. However, the problem occurs with the modelled instructions used in the data dependencies, which may be polymorphic and variant. Polymorphism is not handled effectively in this situation although code normalization may help.

2.5 Classification of Static Program Features

The program features can be divided into four categories of models that enable manipulation of the features suitable for use in detection classification:

- Vectors
- Strings
- Sets
- Graphs

2.5.1 Vectors

Vectors represent the simplest object when processed for classification purposes. Examples of possible vectors in malware classification include opcode distributions [25]. Selecting features and reducing the dimensionality of a vector or feature vector is possible using data mining techniques. Exact matching of vectors can be done quickly, in linear time relative to the dimensionality of the vector. Approximate matching may employ distance metrics or similarity functions. Distance metrics exist between vectors including the Euclidean distance and the Manhattan distance. Additional methods to determine the similarity between two vectors include the cosine similarity.

2.5.2 Strings

Strings are often associated with byte level content in relation to malware classification. Searching for the presence of a substring in a body of text is a traditional technique used in commercial Antivirus. A dictionary search is often used in association with a malware database. The Aho-Corasick [9] string matching algorithm can be performed in a time independent to the size of the database. Extensions to string matching include the use of wildcards in the string, and regular expressions.

Byte level content may be treated as a string and approximate matching performed. The Levenshtein or edit distance between two strings is the minimum number of insertions, deletions and substitutions to transform one string to the other. The edit distance is the basis for an approximate dictionary search which identifies related strings with at most a specific number of errors. Related string metrics to show similarity between strings include the longest common subsequence (LCS), and the sequence alignment algorithms which are used frequently in the Bioinformatics field. The Smith-Waterman algorithm is a widely used for the optimal local sequence alignment.

It is possible to extract all substrings of size n from the text to produce n-grams. Distinct n-grams represent dimensions in a feature vector. This approach can improve the effectiveness and efficiency when performing approximate matching. The use of n-grams also allows for reordering of substrings that the edit distance would penalize heavily. The use of an n-gram feature vector reduces the problem of approximate matching of strings and byte or instruction level content to the problem of approximate vector matching.

Alternative approaches to using strings include the use of statistical or information theory based algorithms to identify measurable properties such as Kolmogorov complexity or entropy.

2.5.3 Sets

A number of malware classification problems are equivalent to showing the similarity between sets or collections of objects. Objects could include the control flow graphs or the basic blocks of a program. An example usage could be to show program similarity by identifying the set similarity between the programs' basic blocks. A number of set similarity functions exist such as the Dice coefficient or the Jaccard index [33].

2.5.4 Graphs

Graphs naturally describe a number of program features including control flow graphs and call graphs. Finding the equivalence between two graphs is to show they are isomorphic. This problem has not been shown to run in polynomial time, but has also not been proven that it does not. Additionally, approximate and inexact graph matching has increased difficulty. Approximate graph matching is based on the graph edit distance or the maximum common subgraph. The graph edit distance is analogous to the string edit distance.

To make graph based classification tractable, a number of approximations have been made. Graphs may be decomposed into subgraphs of fixed sizes where each distinct subgraph represents a feature [28]. The k-subgraph decomposition is analogous to an n-gram decomposition.

2.6 Static Analysis of Malware

Static Analysis is a process of determining properties of an analysed program wherein the program being analysed is not executed. This type of analysis is often employed during program compilation for the purposes of code optimisation. Static Analysis of malware has many benefits in identifying features and building abstract models of malware. These features and models can be used to perform malware classification. Static analysis has been widely investigated, and its scope in this survey limited to its use in malware classification.

2.6.1 Disassembly

Disassembly is the process of translating machine code to assembly language. This is typically the first stage of a static analysis.

Static disassembly parses the entire binary image statically without execution. In static disassembly, there are two main algorithms. In the Linear Sweep algorithm, the instructions are disassembled one instruction after another, starting from the beginning of code. The disadvantage of this method is that data introduced into instruction stream may be erroneously disassembled.

The other main algorithm to perform disassembly is the Recursive Traversal algorithm. This algorithm decodes each instruction following the order of the control flow. This resolves the issue of embedded data, but may miss decoding instructions that are the target of indirect jumps or other situations when it is hard to resolve control flow statically.

Speculative Disassembly attempts to remedy the problems of the Recursive Traversal algorithm problem by first performing the Recursive Traversal, and then performing a Linear Sweep in regions that are not decoded. Christodorescu et al additionally proposed a more robust algorithm in [34] to disassemble binaries that had been purposely obfuscated.

2.6.2 Control Flow Reconstruction

It is necessary to use a program's disassembly to generate inter and intra procedural control flow information. The main hindrance to generating accurate representations is when a program uses indirect branches and procedure calls. The analysis of indirect targets requires data flow analysis. A number of approaches have been employed [35-37], but the simplest approach is to ignore indirect targets completely and accept a less accurate representation. The edges of the graphs representing the control flow can be constructed by connecting the branch or call site to the branch or call target.

2.6.2.1 Opaque Predication Detection

The presence of opaque predicates in a control flow graph reduces the accuracy of the graph because of misleading branch targets. In [38] it was proposed to use the program analysis technique of abstract interpretation to detect specific classes of opaque predicate algorithms.

2.6.3 Alias Analysis of Assembly Language

Alias analysis is an analysis that seeks to statically determine the possible values that pointer variables may contain during program execution. Value-Set Analysis [39] has been proposed as an alias analysis, suitable for binary programs and assembly language. Value-Set Analysis has been used in malware detection [40] and the automated static unpacking of malware [41].

2.6.4 Obfuscation and Limits to Static Analysis

It is known that perfectly precise disassembly is undecidable [32]. Branch targets can be indirect, and precise understanding of those run-time values can be problematic. In [42] an analysis of some limits to static analysis of malware were identified. The use of opaque predicates with hard to analyse predicates were shown to confound the problem of precise program representation. Determining whether two programs are semantically equivalent is also known to an undecidable problem which is why malware detection is often based on heuristic and unsound solutions.

2.7 Automated Unpacking Of Obfuscated Malware

Automated unpacking is the process of revealing the hidden code that is introduced by the code packing transformation. An unpacked binary is important for malware classification because it is required for the static analysis to avoid false classification of the query sample based solely on the packing tool.

2.7.1 Detecting the Code Packing Transformation

It is advantageous to know early in the analysis if a potential malware has undergone a code packing transformation. By knowing that the sample is not packed, further unpacking analysis need not be performed. The process of identifying packed binaries begins with feature extraction. The raw file and section contents can be examined using statistical metrics or machine learning techniques to classify the contents.

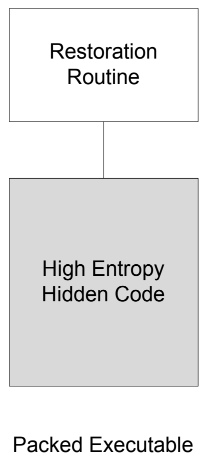


Figure 13. A packed program.

Using entropy analysis to determine if a binary is packed was proposed in Bintropy by Hamrock and Lyda in [43]. The Entropy of a block of data is a statistical measure that describes the amount of information it contains. It is calculated as follows:

$$H(x) = -\sum_{i=1}^{N} \begin{cases} p(i)\log_2 p(i), & p(i) \neq 0 \\ 0, & p(i) = 0 \end{cases}$$

where *p(i)* is the probability of the i^{th} unit of information in event *x*'s sequence of *N* symbols. For the malware packing analysis, the unit of information is a byte value, *N* is 256, and an event is a block of sequential data.

Hamrock and Lyda made the key observation that compressed and encrypted data characterise packed malware samples, and compressed and encrypted data are characterised as having high entropy. Program code and data are found to have much lower entropies. Using this observation, packed malware is identified by the high entropy in its raw content.

Entropy analysis is simple to implement and shown to be effective, yet it has some limitations. Entropy analysis can fail to detect packed malware that intentionally lowers its own entropy. However, this form of evasion is not presently employed by malware. Additionally, entropy analysis can fail to identify code packing transformations which perform simple obfuscations on the malware content, and do not transform and obfuscate the malware using strong encryption or compression. Likewise, code packing that employs instruction virtualization does not require encryption or compression, making entropy analysis unable to identify binaries packed using this method.

2.7.2 Unpacking Using a Dynamic Approach

The majority of research in automated unpacking has targeted code packing transformations that employ a restoration routine. The restoration routine naturally reveals and restores the hidden code. After the restoration routine is complete, the malware transfers control to the restored code. Because the malware naturally reveals the hidden code during execution, dynamic analysis can allow for the extraction of the hidden code and has proven to be popular.

Royal et al proposed an early system employing a combination of static analysis and dynamic analysis in PolyUnpack [13]. A similar technique was proposed in [44]. Polyunpack performed an initial static disassembly of the packed program. During execution, code that became evident and which was not present in the static disassembly, was identified. This was identified as the hidden code. The collection of hidden code constituted the unpacking process. Polyunpack provides a generic solution to unpacking, however performance is not high due to the requirement of disassembling and single stepping through execution. Additionally, the dynamic analysis requires isolation of the running malware. This would imply the use of a virtual machine or whole system emulation with the associated performance cost. This system would not be viable for use in desktop Antivirus.

The most common approach to automated unpacking has taken advantage of the fact that at the original entry point all of the hidden code is revealed. This has resulted in the following components when developing an automated unpacking system.

- Simulation of the malware.
- Detecting when to stop the simulation – when the restoration routine has completed and control is transferred to the original code
- Extraction of the revealed code present in the process image.

Simulation of the malware may involve whole system emulation, hardware based virtualization, or native execution. Execution is simulated until the hidden code is revealed. The most common technique in detecting when to stop the simulation is by maintaining a shadow memory of memory writes, and detecting execution of that memory.

2.7.3 Malware Simulation

2.7.3.1 Whole System Emulation

Renovo was proposed by Kang et al in [21]. Renovo provided a completely dynamic approach to unpacking, employing whole system emulation. The technique of whole system emulation was similarly proposed by Christoderescu et al in [10]. Whole system emulation emulates the physical hardware of a host machine. A complete unmodified guest operating system can be installed on the emulated machine.

Renovo required the use of a kernel driver in the guest operating system being emulated. This is used to track the malware process being executed in the guest system. This requirement of modifying the guest system with a kernel driver may make the system more detectable.

Pandora's Bochs also used whole system emulation, but requires no modifications to the guest operating system, and was proposed by Bohne in [22]. It is similar in concept to Renovo. Renovo utilises a dynamic binary translator based on QEMU [45] to perform the emulation, while Bochs uses an interpreter based emulator. Pandora's Bochs contribution while providing greater resiliency to detection than Renovo is still potentially prone to detection. Attacks to detect whole system emulation were shown in [46]. Methods to respond to these attacks are demonstrated in [47].

Both Pandora's Bochs and Renovo using whole system emulation are quite effective at analysing unknown malware samples if the emulation provides a faithful simulation. However, whole system emulation has shown poor performance. Neither Pandora’s Bochs nor Renovo shows results that are suitable for a real-time Antivirus system.

2.7.3.2 Application Level Emulation

An alternative approach to whole system emulation is to emulate only the operating system interface to guest software. This form of emulation is significantly more efficient because there is no guest operating system that requires execution within the simulation. There has been some commercial interest in application level emulation

[18]. However, little literature has been published and no authoritative refereed publication exists. Likewise, there is almost no evaluation of these systems in existing literature. Application level emulation's main failing is that it provides a less faithful simulation than whole system emulation. This is because the implementer of the emulator must simulate the operating system's operation. In whole system emulation, the installed guest operating provides the authoritative implementation.

2.7.3.3 Dynamic Binary Instrumentation

Dynamic Binary Instrumentation was proposed by Quist in Saffron [48]. Quist proposed instrumenting the malware at runtime to track the execution of dynamically generated code. Saffron employed the use of the DBI framework PIN [49] which has problems with instrumenting anti-debugger code common in malware.

2.7.3.4 Native Execution Hardware Paging

Martignoni et al proposed Omnipack in [50] to natively execute and automatically unpack programs. Hardware page protections were used to monitor the activity of each program. Once unpacked, the image would be scanned by Antivirus software. A similar hardware based approach was employed in [48]. The Omnipack system is implemented to run co-operatively with an operating system, and perform unpacking and virus scanning on demand. The disadvantage of this approach is in the use of the unpacking system on Email gateways, possibly on a different architecture, which forces the provision of a virtual or emulated machine in which to run in. This reduces the level of performance and makes it unsuitable for real-time use.

2.7.3.5 Hardware Based Virtualization

Using hardware based virtualization for malware analysis and automated unpacking was proposed by Dinaburg et al in Ether [51]. In this approach, execution of dynamically generated code triggered extraction of the malware's process image similar to Renovo. The difference is that the simulated environment is provided by a virtual machine using hardware support. Ether, like Pandoras Boch's requires no changes to the guest operating system. Unlike Pandora's Bochs, Ether does not have the same level of problems of a malware detecting the system emulator. However, it has been shown that hardware based virtualization is not immune to detection [46]. The use of a virtual machine, and the use whole system emulation, requires a software license for installation of the guest operating system. This restricts desktop adoption which typically uses a single license. Virtual Machines are also inhibited by slow start-up times which again are problematic for desktop Antivirus use. The use of a Virtual Machine also prevents the system being cross platform as the guest and host CPU's must be of the same architecture.

2.7.4 Detecting End of Unpacking

Detecting when the original entry point is reached and the hidden code of the packed program is revealed allows for subsequent hidden code extraction.

2.7.4.1 Renovo

In Renovo, dynamic code generation was identified by the execution of previously written memory. In this approach, memory is tracked through the maintenance of a shadow memory associated with the running malwares process image.

Malware is executed in the simulated machine and allowed to run until the dynamically generated code is executed. At this point, the memory image of the running malware is taken. There can exist multiple layers or stages of the code packing transformation, so the shadow memory is cleared and the process is restarted. This complete process is reiterated until a time-out expires in any particular stage.

2.7.4.2 Pandora's Bochs

Instead of an exclusive time-out employed by Renovo in each stage to determine when to stop emulation, Pandora's Bochs identified markers that indicate unpacking is still occurring - such indications include if the ratio of memory writes to unique branches is high, the loading of a new dynamic Link Library, executing dynamically generated code, or the first use of dynamically loaded API functions.

2.7.4.3 OmniUnpack

The OmniUnpack approach employed the use of hardware based page protection to monitor writes to memory. Omnipack detects the end of unpacking stage when there is execution of dynamically generated code that invokes a dangerous system call. A dangerous system call is one which can leave the system in an unsafe state. The granularity of tracking memory writes is in the unit of pages. The advantage of the approach employed by Omnipack is that of performance.

2.7.4.4 Uncover

A refinement to the typical technique of detecting execution of dynamically generated code was proposed by Wu et al in [52]. Two additional techniques were used to eliminate false positives. 1) That the stack pointer at the potential original entry point must be the same as when the malware is initially started. 2) That the potential original entry point must constitute part of a sequence of newly or dynamically generated written pages - and those pages must consist of what appears to be code. Determining if a page of memory is code is performed by entropy analysis.

2.7.4.5 Hump-and-dump

Sun et al proposed the Hump-and-Dump [53] method as an alternative for detecting when to stop the simulation. This technique is not based on detecting execution of dynamically generated code. Hump-and-Dump builds a histogram of the ordered addresses of executed instructions. The premise of this technique is to note that the unpacking or restoration routine is evident as a large spike in the histogram. Following the spike, is a flat section of height 1 which normally represents the original entry point. Once the original entry point is detected, simulation ceases and an image of the process is taken to reveal the hidden code. The process can be repeated to account for multiple packing stages. The Hump-and-dump approach requires the use of simulation such as emulation or virtualization.

2.8 Static Approaches to Malware Classification

2.8.1 Classification Approaches

Malware classification is the process of determining if an unknown binary belongs to the class of malicious programs or the class of benign programs.

2.8.1.1 Statistical Classification

A data mining approach to malware detection is to employ statistical classification. Each classification algorithm constructs a model, using machine learning, to represent the benign and malicious classes. In this approach, a labelled training set is required to build the class models during a process of supervised learning. Many statistical classification algorithms exist including Naive Bayes, Neural Networks, and Support Vector Machines. The key to statistical classification is to represent the malicious and benign samples in an appropriate manner to enable the classification algorithms to work effectively. Feature extraction is an important component of effective classification, and an associated feature vector that can accurately represent the invariant characteristics in the training sets and query samples is highly desirable.

2.8.1.2 Instance-Based Learning

Instance-based learning is a related and traditionally popular approach that can be employed wherein the query program is classified by identifying a high similarity to a known instance of malware in the training set. Traditional Antivirus utilises this approach when it performs signature based detection. The key component to perform classification using instance-based learning is a distance or similarity function between

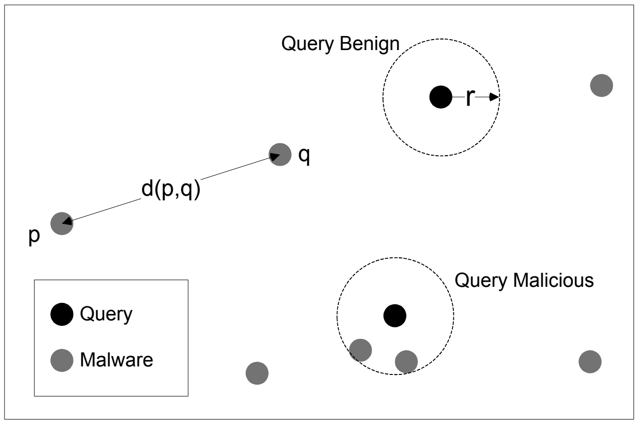


Figure 4. The software similarity search.

the objects representing samples and queries. For a distance function to be effective between objects, the objects must be modelled by a limited set of features that capture the invariant characteristics of the malicious and benign programs. In some cases, the distance function is replaced with a test for equality. However, testing only for equality reduces the effectiveness of the classification process when dealing with malware variants. Instance-based learning can additionally identify high similarity to benign or white-listed samples, depending on the aims of the classification.

2.8.1.3 The Similarity Search Used in Instance-Based Learning

A search of a database to find similar, but not necessarily identical objects to a query is known as a similarity search. The similarity search is a central aspect of instance-based learning when applied to malware detection and classification using a large number of malware signatures and training instances.

Distance functions between objects that have the properties of a metric can employ the use of Metric Access Methods. A similarity search using metric access methods performs faster than exhaustive linear search and enables significantly larger databases without being restricted by an equivalent increase in running time. Metric access methods may use either static [54] or dynamic databases [55]. In dynamic Metric Access Methods, dynamic database operations, such as object insertion, can be effectively performed with reasonable performance expectations.

2.8.2 Control Flow Based Classification Approaches

2.8.2.1 Control Flow Graphs

2.8.2.1.1 Whole Program Control Flow Graph Isomorphism Recognition Using Tree Automata

A fast approach to detecting whole program control flow graph isomorphism and subgraph isomorphism was proposed in [56]. This approach constructed a spanning tree based structure from the control flow graph, and then built a tree automaton for graph recognition. This approach appears to have reasonable performance. However, this technique is not effective at detecting malware variants that alter the control flow or have semantic changes. Nor does this approach attempt to perform unpacking.

2.8.2.1.2 Common k-subgraphs

Decomposing control flow graphs into subgraphs was proposed by Kruegel et al in [28] to classify polymorphic worms. The control flow graphs were decomposed into the set of all subgraphs of fixed size k, where k is the number of nodes in the subpgrah. The k-subgraphs were subsequently transformed into their canonical labelled form. The adjacency matrix of the canonically label graph was transformed into a string. This string represented the k-subgraph feature of the control flow being analysed. Worm

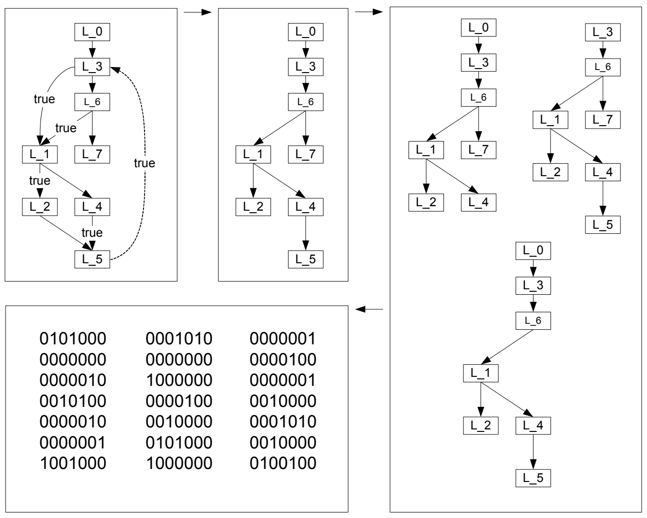


Figure 5. The k-subgraph feature.

detection and classification occured through identifying the prevalence of k-subgraph features between worm like executable content and unclassified executable programs. The performance of this system was reasonable. Because the classification only occurred on network streams identified as potential worms, it is hard to determine the accuracy of the classification when applied to a larger set of malware. Additionally, automated unpacking would be necessary for a general malware classification system.

2.8.2.2 Call Graphs

2.8.2.2.1 Whole Program Context-Free Control Flow

It was proposed in [57] that the inter-procedural control flow information could be represented as a context free grammar with only some loss of information. A string could represent the grammar, and string equality used to show equivalence between the grammar, and inter-procedural control flow they represented. The advantage of this approach, is that string based representations allow for fast searches in a malware database using a dictionary search. The disadvantage of the approach investigated in this research is that it did not employ approximate matching of the inter-procedural control flow. For polymorphic malware variants that alter the control flow through source code modification, an approximate match is necessary for detection of the malware.

2.8.2.2.2 Flowgraph Based Classification using Fixed Points

Carrera proposed an approximate flowgraph matching algorithm in [29] by identifying fixed points in the flowgraphs and successively matching surrounding nodes in the graph. Carrera built a similarity index between malware and used this to build phylogeny (evolutionary) trees for taxonomy. Dullien and Rolles expanded the approximate graph comparison algorithm in [58] to identify identical nodes between callgraphs and control flow graphs. Their algorithm worked by identifying nodes, or fixed points, between binaries that have uniquely identifiable features. Features for a node in the callgraph include the number of basic blocks, control flow edges, and number of subfunction calls. Carrera also proposed an estimation of a control flow graph isomorphism based on string equality and a string signature of the graph representing a graph traversal. Once a set of fixed points were known, their neighbouring nodes could be examined. Identifying neighbours sharing common and unique features iteratively allowed greater parts of the flowgraph to be identified.

The advantage of this approach is that it allows for moderately fast pair-wise comparison between graphs. However, the approach does not perform efficiently for a database of graphs and is not fast enough for desktop Antivirus use. Additionally, automated unpacking, a requirement of the system to perform effectively, was assumed to have occurred before classification is applied. A system for automated unpacking was not proposed.

2.8.2.2.3 Approximating the Graph Edit Distance

An alternative algorithm to approximate graph matching was proposed in the SMIT system [59]. SMIT employed the use of bipartite graphs and the Hungarian algorithm to find matching nodes between two call graphs being compared in $O(N^3)$ running time. The strength of their matching algorithm was that they allowed for it be used as an approximation to the graph edit distance. The graph edit distance between two graphs, is the number of edit operations to convert one graph to the other. The graph edit distance gives a sound basis for similarity and dissimilarity between graphs.

2.8.2.2.4 Metric Access Methods

The graph edit distance is known to have the properties of a metric which allows the use of metric access methods to search a database of objects. The metric access method used in SMIT to perform a nearest neighbor search of call graphs was a Vantage Point Tree [54]. The disadvantage of a Vantage Point Tree is that it is primarily a static data structure. Alternate metric access methods such as the M-Tree [55] can be used for the construction of dynamic structures, allowing for efficient object insertion times.

2.9 Trends

2.9.1 Malware Development

The driving force behind malware development is that of commercial gain by the malware authors. As such, malware development is becoming more rigorous and involves the typical development cycles as seen with legitimate software. Malware creators will continue to protect and extend the lifetime of their software using available techniques at their disposal.

Malware authors have in the past responded to Antivirus detection techniques in an attempt to extend the lifetime of their malware. Techniques were developed for syntactic polymorphism to evade string based signatures. Likewise, dynamic analysis techniques employed by Antivirus systems and researchers including debugging and virtualization are now routinely detected by malware [23, 46]. The detection of individual software systems used for performing analyses will continue. If research systems become popular, it becomes financially rewarding for malware developers to detect these systems. The research community has responded in making analysis systems less identifiable and this trend will continue.

We expect that malware authors will continue to use code packing [13] and polymorphism techniques to obfuscate and hinder analysis. Code packing involving instruction virtualization and malware emulators will grow in use due to the added resistance it provides against malware analysis [14]. Semantic changes to malware will also continue as malware authors reuse already developed malicious code. It is likely that syntactic polymorphism will continue to grow in use. Obfuscations will develop in response to the static analyses used in detection systems to extract features from malware. Incorporating a variety of classification techniques and feature sets can mitigate these attacks.

It should be noted that malware polymorphism development peaked during earlier historical times of virus development. Viruses are now infrequently employed by malware because the motivation for malware development is that of financial gain and not the notoriety once gained for virus writing.

2.9.2 Static Malware Detection and Classification

Malware obfuscation has been increasingly addressed by researchers, and deobfuscation will continue to be developed and incorporated into malware detection systems. These deobfuscation techniques have increasingly borrowed from formal program analyses in an attempt to make sound analyses possible in regards to their given constraints.

Malware classification has employed statistical techniques to detect unknown malware. We believe research will continue using this approach and new features will be developed that can more accurately characterize malware. Instance-based learning will also be developed with particular research opportunity in working with large scale datasets.

Static program features have been extracted at increasing levels of abstraction, and we expect this to continue in future research. Abstraction has the benefit of being resistant to lower level polymorphic changes. The performance of these research systems has not been fully investigated, and we expect that future research opportunity lies in making classification systems practical for industrial and widespread use.

2.10 Summary

Malware is a significant problem in computing environments and has been addressed in research by malware classification systems. Effective malware classification systems must deal with polymorphism. Polymorphic malware introduces syntactic and semantic changes to the malware contents. Traditional byte-level approaches have performed poorly with polymorphic malware. Program abstractions including control flow are observed to be more invariant, when used as static features, than traditional approaches. However, efficient algorithms that use these static features are lacking. Efficiency is a requirement for research systems to be adopted in desktop environments or for the research systems to scale to the high number of malware found in the wild.

The problem of classification and analysis is compounded when a malware is packed and the true contents of the malicious software are hidden. Automated unpacking reveals the hidden content. Efficiency is a key requirement for desktop adoption and widespread use.

3 Problem Definition and Our Approach

The problem of malware classification and variant detection is defined in this chapter. The problem summary is to use instance based learning and perform a similarity search over a malware database. Additionally defined in this chapter is an overview of our approach to design the prototype malware unpacking and classification system, Malwise.

3.1 Problem Definition

A malware classification system is assumed to have advance access to a set of known malware. This is for construction of an initial malware database. The database is constructed by identifying invariant characteristics in each malware and generating an associated signature to be stored in the database. After database initialization, normal use of the system commences. The system has as input a previously unknown binary that is to be classified as being malicious or non malicious. The input binary and the initial malware binaries may have additionally undergone a code packing transformation to hinder static analysis. The classifier calculates similarities between the input binary and each malware in the database. The similarity is measured as a real number between 0 and 1 - 0 indicating not at all similar and 1 indicating an identical or very similar match. This similarity is a based on the similarity between malware characteristics in the database. If the similarity exceeds a given threshold for any malware in the database, then the input binary is deemed a variant of that malware, and

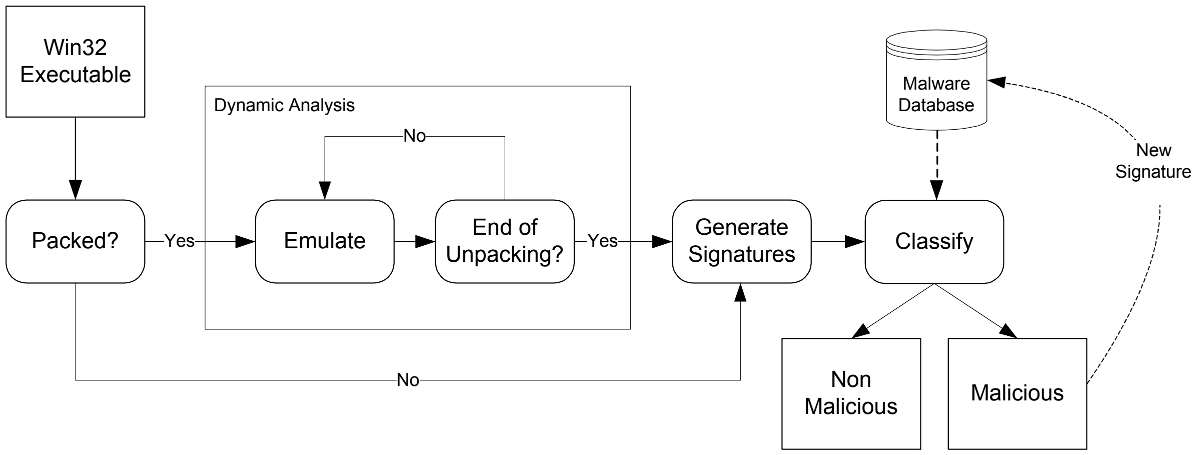


Figure 14. Block diagram of the Malwise malware classification system.

therefore malicious. If identified as a variant, the database may be updated to incorporate the potentially new set of generated signatures associated with that variant.

3.2 Our Approach

Our approach employs both dynamic and static analysis to classify malware. Entropy analysis initially determines if the binary has undergone a code packing transformation. If packed, dynamic analysis employing application level emulation reveals the hidden code using entropy analysis to detect when unpacking is complete. Static analysis then identifies characteristics, building signatures for control flow graphs in each procedure. The similarities between the set of control flow graphs and those in a malware database accumulate to establish a measure of similarity. A similarity search is performed on the malware database to find similar objects to the query. The system design of our prototype system, Malwise, is presented in figure 1. Two approaches are employed to generate and compare flowgraph signatures: exact flowgraph matching and approximate flowgraph matching.

3.2.1 Exact Flowgraph Matching

An ordering of the nodes in the control flow graph is used to generate a string based signature or graph invariant of the flowgraph. String equality between graph invariants is used to estimate isomorphic graphs.

3.2.2 Approximate Flowgraph Matching

The control flow graph is structured in this approach. Structuring is the process of decompiling unstructured control flow into higher level, source code like constructs including structured conditions and iteration. Each signature representing the structured control flow is represented as a string. These signatures are then used for querying the database of known malware using an approximate dictionary search. A similarity between flowgraphs can subsequently be constructed using the string edit distance.

4 Automated Unpacking

Automated unpacking is used to process malware samples before subsequent feature extraction and classification. In this chapter, the automated unpacking component of the Malwise system is proposed and evaluated.

4.1 Identifying Packed Binaries Using Entropy Analysis

Malwise performs an initial analysis on the input binary to determine if it has undergone a code packing transformation. Entropy analysis [43] is used to identify packed binaries. The entropy of a block of data describes the amount of information it contains. Compressed and encrypted data have relatively high entropy. Program code and data have much lower entropy. Packed data is typically characterised as being encrypted or compressed, therefore high entropy in the malware can indicate packing.

An analysis most similar to Uncover [52] is employed. Identification of packed malware is established if there exists sequential blocks of high entropy data in the input binary. If the binary is identified as being packed, then the dynamic analysis to perform automated unpacking proceeds. If the binary is not packed, then the static analysis and classification commences immediately.

4.2 Application Level Emulation

Automated unpacking requires malware execution to be simulated so that the malware may reveal its hidden code. The hidden code once revealed is then extracted from the process image.

Application level emulation provides an alternate approach to whole system emulation for automated unpacking. Application level emulation simulates the instruction set architecture and system call interface. In the Windows OS, the officially supported system call interface is the Windows API.

4.2.1 Interpretation

Malwise utilises interpretation to perform simulation. The features of the emulator implemented by Malwise are described in this section.

4.2.1.1 x86 Instruction Set Architecture (ISA)

Much of the 32-bit x86 ISA has been implemented in Malwise. Extensions to the ISA, including SSE and MMX instructions, have been partially implemented. A partial implementation is adequate because the majority of programs do not employ full use of the ISA. FPU, SSE, and MMX instructions are primarily used by malware to evade or detect emulation. Malware may also use the debugging interface component of the ISA, including debug registers and the trap flag, which are primarily used to obfuscate control flow.

4.2.1.2 Virtual Memory

x86 employs a segmented memory architecture. The Windows OS utilises these segment registers to reference thread specific data. Thread specific data is additionally used by Windows Structured Exception Handling (SEH). SEH is used to gracefully handle abnormal conditions such as division by zero and is routinely used by packers and malware to obfuscate control flow.

Segmented memory is handled in Malwise by maintaining a table of segment descriptions, known in the x86 ISA as the descriptor table. Addressed memory is associated with a segment, known in the ISA as segment selectors, which hold an index into the descriptor table. This enables a translation from segmented addressing to a flat linear addressing.

Virtual memory is maintained by a table of memory regions referenced by their linear address. Each memory region maintains its associated memory contents. Each region also maintains a shadow memory that is utilised by the automated unpacking logic. The shadow memory maintains a flag for each address that is set if that location has been written to or of it has been read.

4.2.1.3 Windows API

The Windows API is the official system call interface provided by Windows. Malwise detects calls to the Windows API by inspecting the simulated program counter. If the program counter contains the address of a Windows API function, then a handler implementing the functionality of the API is executed.

There are too many windows API functions to fully emulate, so only the most common APIs are implemented. Commonly used APIs include heap management, object management, and file system management.

4.2.1.4 Linking and Loading

Program loading entails allocating the appropriate virtual memory, loading the program text, data and dynamic libraries and performing any required relocations. OS specific structures and machine state must also be initialized.

The exported functions of a dynamically linked library may be entirely simulated without having access to the native library. Such a system may have benefit when the emulator is cross platform and when licensing issues should be avoided. Malwise performs full dynamic library loading using the native libraries. This is done to provide a more faithful simulation.

4.2.1.5 Thread and Process Management

Multithreading in applications must be emulated. Malwise implements this using user-level threads - only one thread is running on the host at any particular time and each thread is rescheduled after a specific number of instructions.

Support for emulating multiple processes was not implemented.

4.2.1.6 OS Specific Structures

Windows has process and thread specific structures that require initialization such as the Process Environment Block, Thread Environment Block, and Loader Module. These structures are visible to applications and can be used by malware.

4.2.2 Improvements to Emulation

A naive implementation of emulation can result in poor simulation speed. We make a number of improvements in Malwise as follows, and also make additional improvements to enable a mechanism to address anti-emulation code used by malware.

4.2.2.1 Instruction Predecoding

Instruction predecoding [14] is adopted and produces a significant gain in simulation speed. In this technique, the decoding of unique instructions is cached. This results in a performance gain because disassembly in a naive emulator consumes a large amount of processor time. Predecoding can also be used to cache a function pointer directly to the opcode handler. When used in this way, predecoding allows for fast implementation of the x86 debugging ISA including hardware breakpoints and single step execution used by debuggers. In this optimisation, the cache holding a function pointer to the opcode handler is modified on-demand to reflect that it should execute the breakpoint or trap logic. This removes explicit checks for these conditions from the emulator's main loop.

4.2.2.2 Condition Codes

The x86 condition codes are another point of optimisation and the prototype defers to lazy evaluation of these at the time of their use, similar to QEMU [45].

4.2.2.3 Emulating Known Sections of Code

Many instances of malware use modified variants of the same packer or share similar code between different packers. Taking advantage of this, it is possible to detect known sections of code during emulation and handle them more specifically, and therefore more efficiently than interpretation [60]. To implement this it is noted that each stage during unpacking gives access to a layer of hidden code that has been revealed, and the memory in each layer can be searched for sections of known code. These sections of code can then be emulated, in whole, using custom handlers. This approach achieves significantly greater performance than interpreting each individual instruction. Typical code sections that can have written handlers include decryption loops, decompression loops and checksum calculations. Handlers can also be written and used to dynamically remove specific anti-emulation code.

Malwise implements handlers for frequently used loops in several well known packers.

4.2.3 Verification of Emulation

An automated approach to testing the correctness of emulation is implemented similar to that of testing whole system emulation [61]. To achieve this, the program being emulated is executed in parallel on the host machine. The host program is monitored using the Windows debugging API. At the commencement of each instruction, the emulator machine state is compared against the host version and examined for deviant behaviour. This allows the detection of unfaithful simulation.

Faithful emulation is made more difficult, as some instructions and Windows API functions behave differently when debugged. Malwise rewrites these instructions and functions to emit behaviour consistent to that in a non debugged environment. This enabled testing of packers and malware that employ known techniques to detect and evade debugging.

4.3 Entropy Analysis to Detect Completion of Hidden Code Extraction

Detection of the original entry point (OEP) during emulation identifies the point at which the hidden code is revealed and execution of the original unpacked code begins to take place. Detecting the execution of dynamic code generation by tracking memory writes was used as an estimation of the original entry point in Renovo [21]. In this approach the emulator executes the malware, and a shadow memory is maintained to track newly written memory. If any newly written memory is executed, then the hidden code in the packed binary being will now be revealed. To complicate this approach, multiple layers or stages of hidden code may be present, and malware may be packed more than once. This scenario is handled by clearing the shadow memory contents, continuing emulation, and repeating the monitoring process until a timeout expires.

Malwise extends the concept of identifying the original entry point when unpacking multiple stages by identifying more precisely at which stage to terminate the process, without relying on a timeout. The intuition behind our approach is that if there exists high entropy packed data that has not been used by the packer during execution, then it remains to be unpacked. To determine if a particular stage of unpacking represents the original entry point, the entropy of new or unread memory in the process image is examined. Newly written memory is indicated by the shadow memory for the current stage being unpacked. Unread memory is maintained globally, in a shadow memory for all stages. If the entropy of the analysed data is low, then it is presumed that no more compressed or encrypted data is left to be unpacked. This heuristically indicates completion of unpacking. Malwise also performs the described entropy analysis to detect unpacking completion after a Windows API imposes a significant change to the entropy. This is commonly seen when the packer deallocates large amounts of memory during unpacking. In the remaining case that the original entry point is not identified at any point, an attempt in the emulation to execute an unimplemented Windows API function will have the same effect as having identified the original entry point at this location.

4.4 Discussion

Automated unpacking can potentially be thwarted to result in malware that cannot be unpacked. Application level emulation presents inherent deficiencies when implemented to emulate the Windows operating system. The Windows API is a large set of APIs that requires significant effort to faithfully emulate. Complete emulation of the API has not been achieved in the prototype and faithful emulation of undocumented side effects may be near impossible. Malware that circumvents usual calling mechanisms and malware that employs the use of uncommon APIs may result in incomplete emulation. Malware is reportedly more frequently using the technique of uncommon APIs to evade Antivirus emulation.

An alternative approach is to emulate the Native API which is used by the Windows API implementation. However, the only complete and official documentation for system call interfaces is the Windows API. The Windows API is a library interface, but malware may employ the use of the Native API to interface directly with the kernel.

There does exist reported malware that employ the Native API to evade Antivirus software.

Another problem that exists is early termination of unpacking due to time constraints. Due to real-time constraints of desktop Antivirus, unpacking may be terminated if too much time is consumed during emulation. Malware may employ the use of code which purposely consumes time for the purpose of causing early termination of unpacking. Dynamic binary translation may provide some relief through faster emulation. Additionally, individual cases of anti-emulation code may be treated using custom handlers to perform the simulation where anti-emulation code is detected.

Application level emulation performs optimally against variations of known packers, or unknown packers that do not introduce significantly novel anti-emulation techniques. Many newly discovered malware fulfil these criteria.

4.5 Evaluation

4.5.1 OEP Detection

To verify our system correctly performs hidden code extraction, we tested the Malwise prototype against 14 public packing tools. These tools perform various techniques in the resulting code packing transformation including compression, encryption, code obfuscation, debugger detection and virtual machine detection. The samples chosen to undergo the packing transformation were the Microsoft Windows XP system binaries hostname.exe and calc.exe. hostname.exe is 7680 bytes in size, and calc.exe is 114688 bytes.

The original entry point identified by the unpacking system was compared against what was identified as the real OEP. To identify the real OEP, the program counter was inspected during emulation and the memory at that location examined. If the program counter was found to have the same entry point as the original binary, and the 10 bytes of memory at that location was the same as the original binary, then that address was designated the real OEP.

The results of the OEP detection evaluation are in table 1 and table 2. The revealed code column in the tabulated results identifies the size of the dynamically generated code and data. The number of unpacking stages to reach the real OEP is also tabulated, as is the number of stages actually unpacked using entropy based OEP detection. Finally, the percentage of instructions that were unpacked, compared to the number of instructions that were executed to reach the real OEP is also shown. This last metric is not a definitive metric by itself, as the result of the unaccounted for instructions may not affect the revelation of hidden code – the instructions could be only used for debugger evasion for example. Entries where the OEP was not identified are marked with err. Binaries that failed to pack correctly are marked as fail. The closer the results in column 3 and 4 indicates better performance. The closer the result in column 5 to 100% indicates better performance. A score of 100% indicates a perfect result in unpacking.

Table 1. Metrics for identifying the original entry point in packed samples (hostname.exe).

Name	Revealed code and data	Number of stages to real OEP	Stages unpacked	% of instr. to real OEP unpacked
upx	13107	1	1	100.00
rlpack	6947	1	1	100.00
mew	4808	1	1	100.00
fsg	12348	1	1	100.00
npack	10890	1	1	100.00
expressor	59212	1	1	100.00
packman	10313	2	1	99.99
pe compact	18039	4	3	99.98
acprotect	99900	46	39	98.81
winupack	41250	2	1	98.80
telock	3177	19	15	93.45
yoda's protector	3492	6	2	85.81
aspack	2453	6	1	43.41
pepsin	err	23	err	err

Table 2. Metrics for identifying the original entry point in packed samples (calc.exe).

Name	Revealed code and data	Number of stages to real OEP	Stages unpacked	% of instr. to real OEP unpacked
upx	125308	1	1	100.00
rlpack	114395	1	1	100.00
mew	152822	2	2	100.00
fsg	122936	1	1	100.00
npack	169581	1	1	100.00
expressor	fail	fail	fail	fail
packman	188657	2	1	99.99
pe compact	145239	4	3	99.99
acprotect	251152	209	159	96.51
winupack	143477	2	1	95.84
telock	fail	fail	fail	fail
yoda's protector	112673	6	3	95.82
aspack	227751	4	2	99.90
pespin	err	23	err	err

The results show that unpacking the samples reveals most of the hidden code. The OEP of pespin was not identified, possibly due to unused encrypted data remaining in the process image, which would raise the entropy and affect the heuristic OEP detection. The OEP in the packed calc.exe samples was more accurately identified, relative to the metrics, than in the hostname.exe samples. This may be due to fixed size stages during unpacking that were not executed due to incorrect OEP detection. Interestingly, in many cases, the revealed code was greater than the size of the original unpacked sample. This is because the metric for hidden code is all the code and data that is dynamically generated. Use of the heap, and the dynamic generation of internally used hidden code will increase the resultant amount.

The worst result was in hostname.exe using aspack. 43% of the instructions to the real OEP were not executed, yet nearly 2.5K of hidden of code and data was revealed, which is around a third of the original sample size. While some of this may be heap usage and the result not ideal, it may still potentially result in enough revealed procedures to use for the Malwise classification system in the static analysis phase.

4.5.2 Performance

The system used to evaluate the performance of the unpacking prototype was a modern desktop - a 2.4 GHz Quad core computer, with 4G of memory, running 32-bit Windows Vista Home Premium with Service Pack 1. The performance of the unpacking system is shown in table 3. The running time is total time minus start-up time of 0.60s. Binaries that failed to pack correctly are marked as fail. The number of instructions emulated during unpacking is also shown.

Table 3. Running time to perform unpacking.

	hostname.exe		calc.exe	
Name	Time(s)	# Instr.	Time(s)	# Instr.
mew	0.13	56042	1.21	12691633
fsg	0.13	58138	0.23	964168
upx	0.11	61654	0.19	1008720
packman	0.13	123959	0.28	1999109
npack	0.14	129021	0.40	2604589
aspack	0.15	161183	0.51	4078540
pe compact	0.14	179664	0.83	7691741
expressor	0.20	620932	fail	fail
winupack	0.20	632056	0.93	7889344
yoda’s protector	0.15	659401	0.24	2620100
rlpack	0.18	916590	0.56	7632460
telock	0.20	1304163	fail	fail
acprotect	0.67	3347105	0.53	5364283
pespin	0.64	10482466	1.60	27583453

In this evaluation full interpretation of every instruction is performed. The results demonstrate the system is fast enough for integration into a desktop anti-malware system.

4.6 Summary

The analysis of malicious software is made more challenging due to the presence of packed malware. In this chapter we proposed fast algorithms to unpack malware using application level emulation. We implemented and evaluated a prototype. To detect the completion of unpacking, we proposed and evaluated the use of entropy analysis. The detection of the original entry point worked with a high degree of accuracy. The automated unpacking was demonstrated to work against a promising number of synthetic samples using known packing tools, with high speed. This demonstrated that the automated unpacking system is fast enough for potential desktop integration. The automated unpacking system is efficient and effective and lays the foundation for further malware analysis and classification.

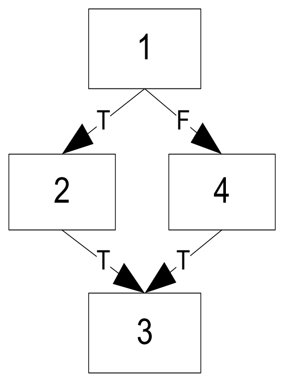
5 Malware Feature Extraction

In this chapter, algorithms to extract the static features of malware are proposed. These features characterize the malware samples and are used for subsequent classification in the Malwise system.

5.1 Static Analysis

The static analysis component of Malwise proceeds once it receives an unpacked binary. The analysis is used to extract characteristics from the input binary that can be used for classification. The characteristic for each procedure in the input binary is obtained through transforming its control flow into compact representation that is amenable to string matching. This transformation, or signature generation, is described in Section 5.2 and 5.3.

To initiate the static analysis process, the memory image of the binary is disassembled using speculative disassembly [34]. Procedures are identified during this stage. A heuristic is used to eliminate incorrectly identified procedures during speculation of disassembly - the target of a call instruction identifies a procedure, only if the call site belongs to an existing procedure. Data runs of more than 256 bytes all having the value of zero are ignored. Once processed, the disassembly is translated into an intermediate representation. Using an intermediate representation is not strictly necessary; however Malwise is built as a general binary analysis platform which utilizes the intermediate form. The intermediate representation is used to generate an architecture independent control flow graph for each identified procedure. The control flow graph is then



```
(1 -> 2), (1 -> 4)
(2 -> 3), ()
(), ()
(4 -> 3), ()
```

Figure 15. A depth first ordered flowgraph and its signature.

transformed into a signature represented as a character string. The signature is also associated with a weight, described in the following sections. The weight intuitively represents the importance of the signature when used to determine program similarity.

5.2 Exact Flowgraph Matching

It is possible to generate a signature using a fast and simple method if the matching algorithm only identifies graph isomorphism [29]. This approach takes note that if the signatures or graph invariants of two graphs are not the same, then the graphs are not isomorphic. The converse, while not strictly sound, is used as a good estimate to indicate isomorphism. To generate a signature, the algorithm orders the nodes in the control flow graph using a depth first order, although other orderings are equally sufficient. A signature subsequently consists of a list of graph edges for the ordered nodes, using the node ordering as node labels. This signature can be represented as a string. An example signature is shown in figure 15.

To improve the performance, a hash of the string signature can be used instead. CRC64 is used in Malwise. The advantage of this matching algorithm over approximate matching is that classification using exact matches of signatures can be performed very efficiently using a dictionary lookup.

The normalized weight of procedure *x* is defined as:

$$weight_x = \frac{B_x}{\sum_i B_i}$$

where B_i is the number of basic blocks of procedure *i* in the binary.

The similarity ratio between two flowgraphs in exact matching, with signatures *x* and *y* is:

$$w_{ed} = \begin{cases} 1, & x = y \\ 0, & x \neq y \end{cases}$$

In Malwise, balanced binary trees implement the exact search of the flowgraph database. The runtime complexity is O(log(N)).

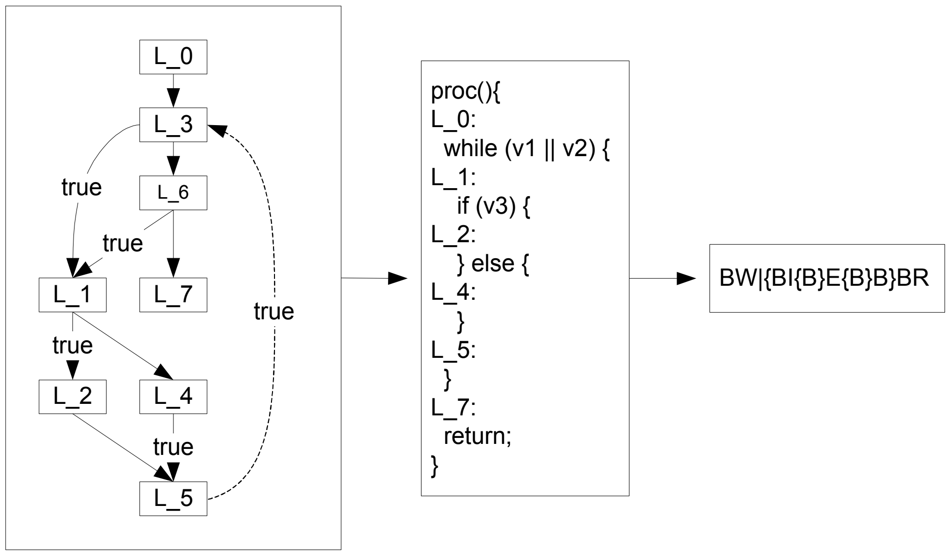
5.3 Approximate Flowgraph Matching

Malware classification using approximate matches of signatures is employed. Intuitively, using approximate matches of a control flow graph, instead of exact isomorphism tests, should enable identification a greater number of malware variants. In our approach we use structuring to generate a signature that enables approximate matching using string edit distances.

Structuring is the process of recovering high level structured control flow from a control flow graph. In our system, the control flow graphs in a binary are structured to produce signatures that are amenable to comparison and approximate matching using string edit distances.

The intuition behind using structuring as a signature is that similarities between malware variants are reflected by variants sharing similar high level structured control flow. If the source code of the variant is a modified version of the original malware, then this intuition would appear to hold true.

The structuring algorithm implemented in Malwise is a modified algorithm of that proposed in the DCC decompiler [62]. If the algorithm cannot structure the control flow graph then an unstructured branch is generated. Surprisingly, even when graphs are reducible (a measure of how inherently structured the graph is), the algorithm generates unstructured branches in a small but not insignificant number of cases. Further



```
proc(){
L_0:
  while (v1 || v2) {
L_1:
    if (v3) {
L_2:
    } else {
L_4:
    }
L_5:
  }
L_7:
  return;
}
```

```
BW|{BI{B}E{B}B}BR
```

Figure 16. The relationship between a control flow graph, a high level structured graph, and a signature.

improvements to this algorithm to reduce the generation of unstructured branches have been proposed [63, 64]. However, these improvements were not implemented.

The result of structuring is output consisting of a string of character tokens representing high level structured constructs that are typical in a structured programming language. Subfunction calls are represented, as are gotos; however, the goto and subfunction targets are ignored. The grammar for a resulting signature is defined in figure 17.

The normalized weight of procedure *x* is defined as:

$$weight_x = \frac{len(s_x)}{\sum_i len(s_i)}$$

where s_i is signature of procedure *i* in the binary. The weights are normalized so that the sum of the set of weights is equal to 1.

The similarity ratio [26] was proposed to measure the similarity between basic blocks. It is used in our research to establish the number of allowable errors between flowgraph signatures in an approximate dictionary search. For two signatures or structured graphs represented as strings *x* and *y*, the similarity ratio is defined as:

$$w_{ed} = 1 - \frac{ed(x,y)}{\max(len(x), len(y))}$$

where *ed(x,y)* is the edit distance. Malwise defines the edit distance as the Levenshtein distance – the number of insertions, deletions, and substitutions to convert one string to another. Signatures that have a similarity ratio equal or exceeding a threshold *t* (*t*=0.9) are identified as positive matches. This figure was derived empirically through a pilot study.

```
Procedure            ::= StatementList

StatementList        ::= Statement | Statement StatementList

Statement            ::= Return | Break | Continue | Goto | Conditional | Loop | BasicBlock

Goto                 ::= 'G'

Return               ::= 'R'

Break                ::= 'B'

Continue             ::= 'C'

BasicBlock           ::= 'B' | 'B' SubRoutineList

SubRoutineList       ::= 'S' | 'S' SubRoutineList

Condition            ::= | ConditionTerm | ConditionTerm NextConditionTerm

NextConditionTerm    ::= '!' Condition | Condition

ConditionTerm        ::= '&' | '|'

IfThenCondition      ::= Condition | '!' Condition

Conditional          ::= IfThen | IfThenElse

IfThen               ::= 'I' IfThenCondition '{' StatementList '}'

IfThenElse           ::=  'I' Condition '{' StatementList ‘}’ ‘E’ ‘{‘ StatementList '}'

Loop                 ::= PreTestedLoop | PostTestedLoop | EndlessLoop

PreTestedLoop        ::= 'W' Condition '{' StatementList '}'

PostTestedLoop       ::= 'D' '{' StatementList '}' Condition

EndlessLoop          ::= 'F' '{' StatementList '}'
```

Figure 17. The grammar to represent a structured control flow graph signature.

Using the similarity ratio *t* as a threshold, the number of allowable errors, *E*, or edit distance, for signature x to be identified as a matching graph, is defined as:

$$E = len(x)(1-t)$$

To identify matching graphs from a flowgraph database, an approximate dictionary search is performed on signature *x*, allowing *E* errors. The search is performed using BK Trees [65]. BK Trees exploit knowledge that the Levenshtein distance forms a metric space. The BK Tree search algorithm is faster than an exhaustive comparison of each signature in the dictionary.

The runtime complexity of the edit distance between two signatures or strings is O(nm), where n and m are the lengths of each respective signature. The algorithm employs dynamic programming.

5.4 Discussion

Malware classification based on static analysis has a number of inherent problems and may fail to perform correctly in all cases. Performing static disassembly, identifying procedures and generating control flow graphs is, in the general case, undecidable. Malware may specifically craft itself to make static analysis hard. In practice, the majority of malware is compiled from a high level language and obfuscated as a post-processing stage. The primary method of obfuscation is the code packing transformation. Due to these considerations, static analysis generally performs well in practice.

5.5 Summary

Malware variants can be detected by identifying similarity in control flow to existing malware. In this chapter we proposed two algorithms to extract control flow graph features from malware. We proposed an algorithm using an estimation of control flow graph isomorphism through the string equality of graph invariants. We also proposed the decompilation technique of structuring to generate a string signature of a flowgraph for use in approximate graph matching. The structured signature was amenable to approximate matching using the string edit distance. These features lay the foundation for malware classification.

6 Malware Classification

6.1 Malware Classification Using Set Similarity

To classify an input binary, the analysis makes use of a malware database. The database contains the sets of flowgraph signatures, represented as strings, of known malware. To classify the input binary, a similarity is constructed between the set of the binary's flowgraph strings and each set of flowgraphs associated with malware in the database.

To construct the similarity between the two sets of flowgraph strings we construct a mapping or assignment between the strings from each set. For exact matching, the assignment is based on string equality. For approximate matching, a greedy assignment is made for the best approximate matching string where the similarity ratio is above 0.9. An example assignment is shown in figure 18.

Two weights are associated with each matching flowgraph signature. The weights have been normalized and the sum of matching weights identifies the size of the matching subset. Formally, the asymmetric similarity is:

$$S_x = \sum_i \begin{cases} 0, & w_{ed_i} < t \\ w_{ed_i} weight_{x_i}, & w_{ed_i} \geq t \end{cases}$$

where t is the empirical threshold value of 0.9, w_{ed} is the similarity ratio between the i^{th} control flow graph of the input binary and the matching graph in the malware database, and $weight_x$ is the weight of the cfg where x is either the input binary or the malware binary in the database.

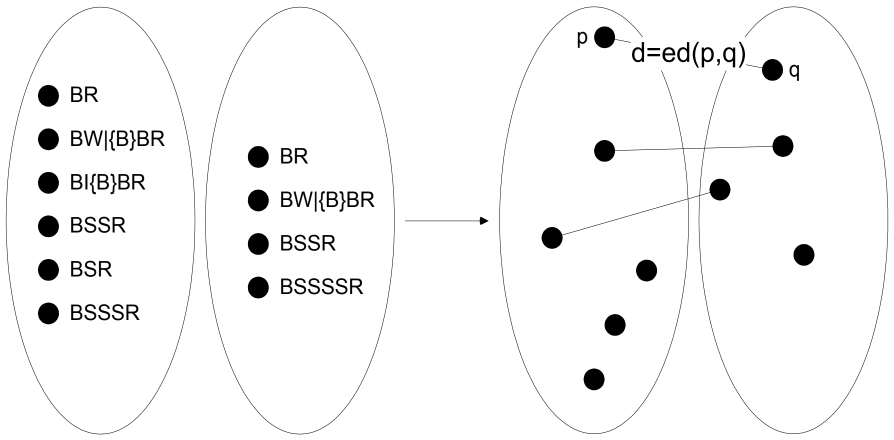


Figure 18. Assignment of flowgraph strings between sets.

The analysis performs more accurately with a greater number of procedures and hence signatures. If the input binary has too few procedures, then classification cannot be performed. The prototype does not perform classification on binaries with less than 10 procedures. For the exact matching classification, an additional requirement is that the control flow graph has at least 5 basic blocks.

The program similarity is the final measure of similarity used for classification and is the product of the asymmetric similarities. The program similarity is defined as:

$$S(i,d) = S_i S_d$$

where i is the input binary, d is the database malware instance, S_i and S_d are the asymmetric symmetries. An example construction of program similarity is shown in figure 19.

If the program similarity of the examined program to any malware in the database equals or exceeds a threshold of 0.6, then it is deemed to be a variant. As the database contains only malicious software, the binary of unknown status is also deemed

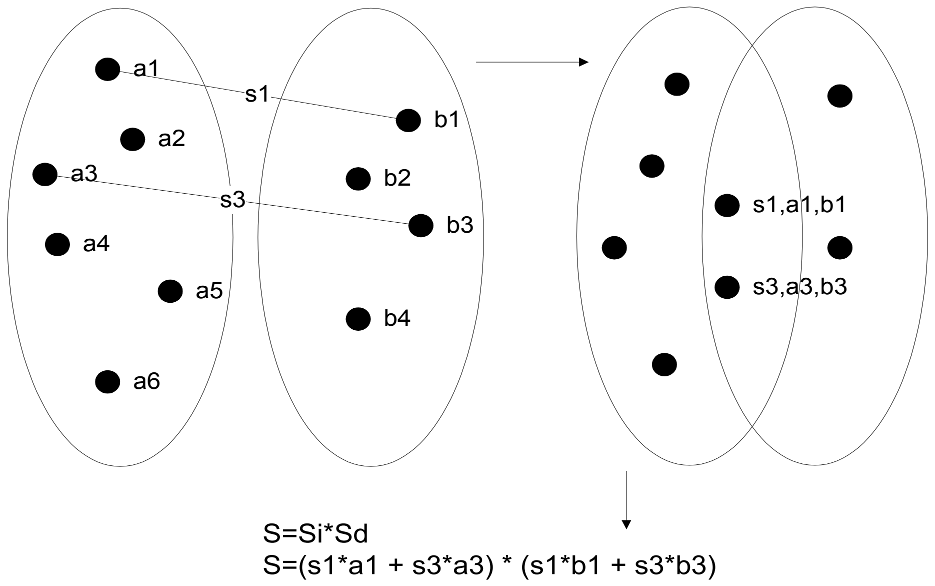


Figure 19. Malware classification using set similarity.

malicious. The threshold of 0.6 was chosen empirically through a pilot study. If the binary is identified as malicious, and not deemed as excessively similar to an existing malware in the database, the new set of malware signatures can be stored in the database as part of an automatic system. Program similarity exceeding 0.95 is used in Malwise to define signatures excessively similar.

6.2 The Set Similarity Search

To classify the query program as malicious or benign, a similarity search is performed to find any similar malware in the database. The search can be performed exhaustively but has poor performance. To improve the performance, the similarity between programs, represented as sets, can utilise an alternative algorithm. The expected case when performing the set similarity search, is that the query is not similar to any malware in the database and our algorithm exploits this expected case.

Our first proposed algorithm iterates through each flowgraph string in the query program and finds matching strings from malware using a global database. From this, the asymmetric similarities associated with each malware are constructed during each round. After processing the query program, the matching malware are examined to identify those that have a program similarity above the threshold of 0.6.

The problem with this initial approach is that some flowgraph strings have many matching malware. To handle this problem, we divide the classification process into two stages. In the first stage, we only build the asymmetric similarity for flowgraphs which are associated with a unique or nearly unique malware. At completion of processing uniquely matching malware, we prune those that cannot have an eventual program similarity above 0.6. Finally, we process the remaining flowgraph strings, but we do not employ the entire flowgraph database, and instead use a local database for each of the malware remaining from the previous stage. Pseudo code to describe the algorithm is given in figure 20. We then return the remaining malware equal to or exceeding the program similarity of 0.6. This part of the process is not shown to conserve space.

The set similarity search algorithm can be used for approximate matching by using an approximate dictionary search over the standard dictionary search used in exact matching. The similarity ratio threshold defines the maximum number of errors allowed in the search.

```
S = 0.6
matches[name][Sa,Sb]   : output       : input initialized Sa=0, Sb=0
db                     : input        : malware database
in                     : input        : input binary
solutions              : global temporary

ProcessMatch(s: malware signature, similarityTogo)
{
        if (!seenBefore(s) && !solutions.seenBefore(s.malwareName)) {
                if (!matches[s.malwareName].find(s) and similarityTogo < S) {
                        // do nothing
                } else if (matches.find(s) &&
                        similarityTogo + matches[s.malwareName].Sa < S &&
                        similarityTogo + matches[s.malwareName].Sb < S)
                {
                        matches[s.malwareName].erase(s)
                } else {
                        matches[s.malwareName].Sa += weight_of_malware_cfg(s)
                        matches[s.malwareName].Sb += weight_of_input_cfg(s)
                }
        }
}

Classify(in: input binary, db: malware database)
{
        similarityTogo = 1.0
        foreach u in unique_cfg_matches(db, cfgs(in)) {
                solutions.reset()
                ProcessMatch(u, similarityTogo)
                similarityTogo -= weight_of_input_cfg(u)
        }
        dups = duplicate_cfg_matches(db, cfgs(in))
        foreach d in dups {
                if (1.0 - similarityTogo >= 1.0 - S)
                        break
                solutions.reset()
                foreach e in cfgs(d) {
                        ProcessMatch(malware_signature(d), similarityTogo)
                }
                similarityTogo -= weight_of_input_cfg(u)
                dups.erase(d)
        }
        foreach c in matches {
                tempSimilarityTogo = similarityTogo
                foreach d in dups {
                        solutions.reset()
                        foreach e in matching_cfgs_in_specific_db(db, d, c.malwareName)) {
                                ProcessMatch(malware_signature(e), tempSimilarityTogo)
                        }
                        tempSimilarityTogo -= weight_of_input_cfg(d)
                }
        }
        return matches
}
```

Figure 20. Pseudo code for the set similarity search.

6.3 Complexity Analysis

We assume a search complexity is O(log(N)) for both global and local flowgraph databases. The runtime complexity of malware classification is on average O(Nlog(M)) where M is the number of control flow graphs in the database, and N is the number of control flow graphs in the input binary. N is proportional to the input binary size and not more than several hundred in most cases. The worst case can be expected to have a runtime complexity of O(Nlog(M) + ANlog(N)), where A is the number of similar malware to the input binary. It is desirable that the malware database is not populated with a significant number of similar malware. In practice, this condition is unlikely to be significant. It is expected that the average case is processing benign samples.

The runtime complexity, in existing literature, to identify similarity between two call graphs using the Hungarian method [59] is N^3, where N is the sum of nodes in each graph. Metric trees can avoid exhaustive comparisons in the database, which naively would be MN^3, where M is the number of indexed malware. An average of 70% of the database size M, was pruned when identifying the 10 nearest neighbours in a search utilizing metric trees [59]. Our algorithm, has similar intentions and comparable results in identifying malware variants, and performs significantly more efficiently. The runtime complexity of a typical multi-pattern string matching algorithm used in Antivirus systems, employing the Aho-Corasick algorithm [9] is linear to the size of the input program and number of identified matches. The disadvantage of this approach is that pre-processing is required on the malware database to enable linear scanning time that is independent of the database size. Our system imposes more overhead by performing unpacking and static analysis, but is potentially capable of real-time updates to the malware database, and is capable of maintaining efficient runtime complexity. Additionally, in traditional Antivirus, false positives increase as the program sizes increase [56]. Our system is more resilient to false positives under these conditions because increased flowgraph complexity enables more precise signatures.

6.4 Evaluation

6.4.1 Effectiveness

To compare the effectiveness of exact matching and approximate matching, 40 malware variants from the Netsky, Klez, Roron and Frethem families of malware were classified. The Netsky, Klez and Roron malware samples were chosen to mimic a selection of the malware and evaluation metrics in previous research [29]. The malware was obtained through a public database [66]. A number of the malware samples were packed. Malwise automatically identifies and unpacks such malware as necessary. Each of the 40 malware sample were compared to every other sample. In approximate matching, 252 comparisons identified variants. The same evaluation was performed using exact matching, and 188 comparisons identified variants. Approximate matching identifies more variants as expected. Exact matching, while less accurate, is demonstrated to be effective at detecting malware variants.

Table 5. Similarity matrices for malware using exact matching.

	a	b	c	d	g	h
a		0.76	0.82	0.69	0.52	0.51
b	0.76		0.83	0.80	0.52	0.51
c	0.82	0.83		0.69	0.51	0.51
d	0.69	0.80	0.69		0.51	0.50
g	0.52	0.52	0.51	0.51		0.85
h	0.51	0.51	0.51	0.50	0.85	

Klez (exact).

	aa	ac	f	j	p	t	x	y
aa		0.74	0.59	0.67	0.49	0.72	0.50	0.83
ac	0.74		0.69	0.78	0.40	0.55	0.37	0.63
f	0.59	0.69		0.88	0.44	0.61	0.41	0.70
j	0.67	0.78	0.88		0.49	0.69	0.46	0.79
p	0.49	0.40	0.44	0.49		0.68	0.85	0.58
t	0.72	0.55	0.61	0.69	0.68		0.63	0.86
x	0.50	0.37	0.41	0.46	0.85	0.63		0.54
y	0.83	0.63	0.70	0.79	0.58	0.86	0.54	

Netsky(exact).

	ao	b	d	e	g	k	m	q	a
ao		0.44	0.28	0.27	0.28	0.55	0.44	0.44	0.47
b	0.44		0.27	0.27	0.27	0.51	1.00	1.00	0.58
d	0.28	0.27		0.48	0.56	0.27	0.27	0.27	0.27
e	0.27	0.27	0.48		0.59	0.27	0.27	0.27	0.27
g	0.28	0.27	0.56	0.59		0.27	0.27	0.27	0.27
k	0.55	0.51	0.27	0.27	0.27		0.51	0.51	0.75
m	0.44	1.00	0.27	0.27	0.27	0.51		1.00	0.58
q	0.44	1.00	0.27	0.27	0.27	0.51	1.00		0.58
a	0.47	0.58	0.27	0.27	0.27	0.75	0.58	0.58	

Roron (exact).

Table 6. Rorom malware and similarity ratio threshold of 1.0.

	ao	b	d	e	g	k	m	q	a
ao		0.41	0.27	0.27	0.27	0.46	0.41	0.41	0.44
b	0.41		0.27	0.26	0.27	0.48	1.00	1.00	0.56
d	0.27	0.27		0.44	0.50	0.27	0.27	0.27	0.27
e	0.27	0.26	0.44		0.56	0.26	0.26	0.26	0.26
g	0.27	0.27	0.50	0.56		0.26	0.27	0.27	0.26
k	0.46	0.48	0.27	0.26	0.26		0.48	0.48	0.73
m	0.41	1.00	0.27	0.26	0.27	0.48		1.00	0.56
q	0.41	1.00	0.27	0.26	0.27	0.48	1.00		0.56
a	0.44	0.56	0.27	0.26	0.26	0.73	0.56	0.56	

Table 4. Similarity matrices for malware using approximate matching.

	a	b	c	d	g	h
a		0.84	1.00	0.76	0.47	0.47
b	0.84		0.84	0.87	0.46	0.46
c	1.00	0.84		0.76	0.47	0.47
d	0.76	0.87	0.76		0.46	0.45
g	0.47	0.46	0.47	0.46		0.83
h	0.47	0.46	0.47	0.45	0.83	

Klez (approximate).

	aa	ac	f	j	p	t	x	y
aa		0.78	0.61	0.70	0.47	0.67	0.44	0.81
ac	0.78		0.66	0.75	0.41	0.53	0.35	0.64
f	0.61	0.66		0.86	0.46	0.59	0.39	0.72
j	0.70	0.75	0.86		0.52	0.67	0.44	0.83
p	0.47	0.41	0.46	0.52		0.61	0.79	0.56
t	0.67	0.53	0.59	0.67	0.61		0.61	0.79
x	0.44	0.35	0.39	0.44	0.79	0.61		0.49
y	0.81	0.64	0.72	0.83	0.56	0.79	0.49	

Netsky (approximate).

	ao	b	d	e	g	k	m	q	a
ao		0.70	0.28	0.28	0.27	0.75	0.70	0.70	0.75
b	0.74		0.31	0.34	0.33	0.82	1.00	1.00	0.87
d	0.28	0.29		0.50	0.74	0.29	0.29	0.29	0.29
e	0.31	0.34	0.50		0.64	0.32	0.34	0.34	0.33
g	0.27	0.33	0.74	0.64		0.29	0.33	0.33	0.30
k	0.75	0.82	0.29	0.30	0.29		0.82	0.82	0.96
m	0.74	1.00	0.31	0.34	0.33	0.82		1.00	0.87
q	0.74	1.00	0.31	0.34	0.33	0.82	1.00		0.87
a	0.75	0.87	0.30	0.31	0.30	0.96	0.87	0.87	

Roron (approximate).

Table 4 and table 6 evaluates the flowgraph matching system in more detail using generated similarities between malware using approximate and exact matching. In normal operation, the system does not calculate the complete similarity between binaries which are not considered variants, however this performance feature was relaxed for this evaluation metric. Highlighted cells identify a malware variant, defined as having a similarity equal to or exceeding 0.60. In approximate matching, a flowgraph is classed as being a variant of another flowgraph if the similarity ratio is equal to or in excess of 0.9. To improve the performance of exact matching, procedures with less than 5 basic blocks were not included, which on occasion results in higher similarity being identified than approximate matching, as demonstrated by the Netsky.t and Netsky.f malware. The results demonstrate that the system finds high similarities between malware families using both approximate and exact matching.

Table 5 shows the difference in the similarity matrix when the threshold for the similarity ratio is increased to 1.0. Differences of up to 30% were noted across the malware variants using the two similarity ratio thresholds. Using a threshold of 1.0 for the similarity ratio is similar, but not identical, to the results of exact matching.

6.4.2 Effectiveness of Exact Matching

To evaluate exact matching in Malwise on a larger scale, 15,409 malware samples with unique MD5 hashes were collected between 02-01-2009 and 8-12-2009 from honeypots in the mwcollect Alliance [67] network. The malware samples were sorted according to collection time, and processed in order. 94.4% of malware samples were found to have a similarity of more than 95% to previously classified malware in the set. 863

representative malware signatures were stored in the database, where none were more than 95% similar to other signatures. It was found that 88.26% of malware were detected as variants of previously classified malware. This high probability represents strong evidence that detecting malware variants has much benefit in the detection of unknown malware samples. It was also found that 34.24% of malware were 100% similar to existing malware, once unpacked. This corroborates research [16] that many new instances of malware are repacked versions of existing malware. The results after evaluating 15,409 malware, demonstrate the classification algorithm used by Malwise is highly effective in detecting malware. The accuracy of these results is dependent on successfully unpacking the malware samples. Manual inspection was performed on a smaller set of samples shown in Section 6.4.3 to validate the results.

6.4.3 Efficiency of Exact Matching

809 malware samples with unique MD5 hashes were collected between 29-04-2009 and 17-05-2009 from honeypots in the mwcollect Alliance network [67] and form a subset of the previously classified 15,409 malware. All malware were used to populate the database, irrespective of having identical or near identical signatures to existing malware. 754 samples were found to have at least one other sample in the set which was a variant. Table 7 and figure 21 evaluates the speed of processing these malware samples, including unpacking and classification time but excluding the loading time of the malware database. The evaluation was performed on a 2.4 GHz Quad Core Desktop PC with 4G of memory, running 32-bit Windows Vista Home Premium with Service Pack 1, as was used in the unpacking performance testing. 86% of the malware were processed in under 1.3 seconds. The only malware that was not processed in under 5

Table 8. Benign sample processing time.

Time(s)	Num. of Samples
0.0	0
0.1	139
0.2	80
0.3	42
0.4	28
0.5	10
0.6	10
0.7	3
0.8	6
0.9	5
1-2	17
2+	6

Table 7. Malware processing time.

Time(s)	Num. of Samples
0-1	299
1-2	401
2-3	46
3-4	30
4-5	32
5+	1

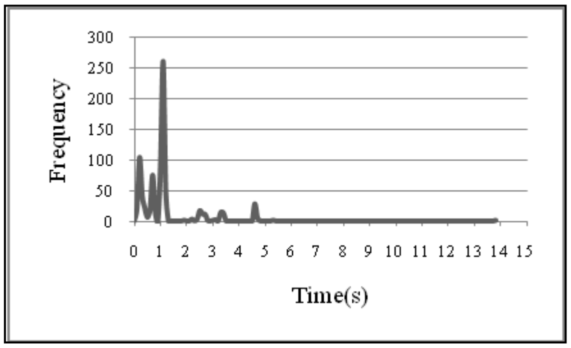


Figure 21. Malware processing time.

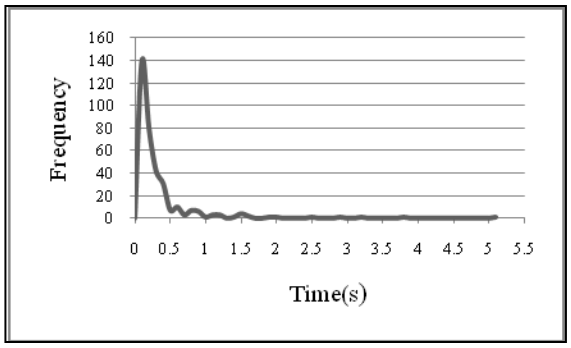


Figure 22. Benign processing time.

seconds instead took nearly 14 seconds. This was because nearly 163 Million instructions were emulated during unpacking. This is possibly the result of an anti-emulation loop. Manual inspection of the results also reveal some malware were not fully unpacked. The static analysis is therefore likely generating signatures based on the packing tool, which becomes blacklisted by system.

To evaluate the speed of classifying benign samples, 346 binaries in the Windows system directory were evaluated using the malware database created in the previous evaluation. The results are shown in table 8 and figure 22. The median time to perform classification was 0.25 seconds. The slowest sample classified required 5.12 seconds. Only 6 samples required more than 2 seconds.

It is much faster to process benign samples than malicious samples. Malicious samples are typically packed and the unpacking consumes the majority of processing time. The results clearly show this difference, and give more evidence that our system performs quickly in the average case. The results shown demonstrate efficient processing in the majority of benign and real malware samples, with speeds suitable for potential desktop adoption.

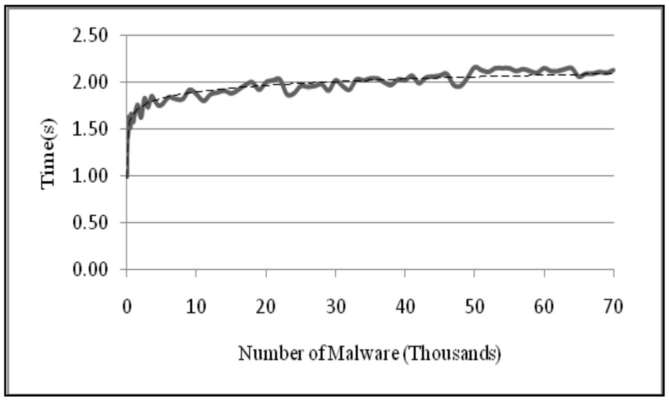


Figure 23. Scalability of classification.

6.4.4 Efficiency of Exact Matching With A Synthetic Database

To evaluate the scalability of the classification algorithm used in exact matching, a synthetic database was constructed. To simulate conditions likely in real samples, 10% of the control flow graphs were made common to all malware. The synthetic database contained up to a maximum of 70,000 malware, with each malware having 200 control flow graphs. The malware signatures were randomly generated. The time to perform 100,000 repetitions of classification of an executable and no other processing is shown in figure 23. Less than a millisecond was required to complete a single repetition of classification for all evaluated database sizes. The trend of the graph is logarithmic, as predicted, when classifying a benign binary.

Table 9. Similarity matrix for non similar programs using approximate matching.

	cmd.exe	calc.exe	netsky.aa	klez.a	roron.ao
cmd.exe		0.00	0.00	0.00	0.00
calc.exe	0.00		0.00	0.00	0.00
netsky.aa	0.00	0.00		0.19	0.08
klez.a	0.00	0.00	0.19		0.15
roron.ao	0.00	0.00	0.08	0.15	

Table 10. Similarity matrix for non similar programs using exact matching.

	cmd.exe	calc.exe	netsky.aa	klez.a	roron.ao
cmd.exe		0.00	0.00		0.00
calc.exe	0.00		0.00	0.00	0.00
netsky.aa	0.00	0.00		0.15	0.09
klez.a		0.00	0.15		0.13
roron.ao	0.00	0.00	0.09	0.13	

6.4.5 Malwise's Resilience to False Positives

To evaluate the generation of false positives in Malwise, table 9 and table 10 shows classification among non similar binaries using approximate and exact matching. Low similarity was found among these samples as expected.

To further evaluate the exact matching algorithm against false positives, the malware database created from the 809 samples in Section 6.4.3 was used for classifying the binaries in the windows system directory. No false positives were identified. The highest matching sample showed a similarity of 0.34. All other binaries had similarities below 0.25. This result clearly shows resilience against false positives.

Table 11. Histogram of similarities between executable files in Windows system directory.

Similarity	Matches (approx.)	Matches (exact)
0.0	105497	97791
0.1	2268	1598
0.2	637	532
0.3	342	324
0.4	199	175
0.5	121	122
0.6	44	34
0.7	72	24
0.8	24	22
0.9	20	12
1.0	6	0

To continue evaluation of exact and approximate matching, table 11 shows a more thorough test for false positive generation by comparing each executable binary to every other binary in the Windows Vista system directory. The histogram groups binaries that shares similarity in buckets grouped in intervals of 0.1. The results show there exist similarities between some of the binaries, but for the majority of comparisons the similarity is less than 0.1. This seems a reasonable result as most binaries will be unrelated. Exact matching identifies fewer similarities than approximate matching as expected. Exact matching also produces fewer comparisons due to the added requirement of each flowgraph having at least 5 basic blocks, which resulted in some binaries being ineligible for analysis.

6.5 Summary

Malware can be classified according to similarity in its flowgraphs. We proposed an algorithm to identify the similarity between programs based on sets of control flow graph features. We additionally proposed a similarity search algorithm that allowed for efficient database searching to find similar sets to our query. We implemented these algorithms in the prototype Malwise system. It was shown that our system can effectively identify variants of malware in samples of real malware. It was also shown that there is a high probability that new malware is a variant of existing malware. Finally, we evaluated the speed and efficiency of the complete Malwise system including unpacking and malware classification. The demonstrated speed warrants Malwise as suitable for potential applications including desktop and Internet gateway and Antivirus systems.

7 Conclusions and Future Work

7.1 Future Work

The Malwise system performs effectively but we believe the malware detection rate could be improved by employing more precise algorithms when comparing and assigning control flow graphs between sets of programs. The Malwise system currently employs a greedy solution to the assignment problem. This could be replaced with an optimal assignment to minimize the sum of distances.

In addition to effectiveness, the efficiency of the Malwise system could also potentially be improved. The automated unpacking system could employ dynamic binary translation. Approximate matching could use heuristic based comparisons. The more sound string edit distance could subsequently be used to refine the results. Additionally, alternative string metrics are possible such as the sequence alignment algorithms frequently employed in the field of Bioinformatics.

The malware detection could also be made more robust against different forms of polymorphism. Particular features may be found to be more effective in particular situations. The use of multiple features, including call graph information and data dependencies, could be used. Finally, statistical classification could be applied to control flow features in the detection of unknown and novel malware samples.

7.2 Conclusions

This thesis provided a survey of existing literature in the automated unpacking of malware and static classification of malware. The thesis proposed novel approaches to effectively unpack and classify malware while maintaining a high degree of efficiency. Our approach employed application level emulation for unpacking malware and used control flow graphs as static features to characterize malware.

The major contributions of this thesis are summarized follows:

- We proposed the use of application level emulation for automated unpacking.
- We proposed using entropy analysis to detect when unpacking was complete.
- We proposed using a graph invariant based signature to estimate control flow graph isomorphism for the purpose of constructing a measure of program similarity.
- We proposed using the decompilation technique of structuring to generate a string based control flow signature, amenable to comparisons using the string edit distance. This approach was used for approximate control flow graph matching.
- We proposed a set similarity function and a set similarity search algorithm which formed the basis for our malware classification system and performed efficiently in the expected case.

We implemented and evaluated our ideas in a novel prototype system named Malwise. The automated unpacking system was found to accurately unpack samples that were obfuscated using known packing tools. The speed and efficiency of the unpacking system was found to be suitable for potential desktop adoption. The malware classification system was demonstrated to detect variants of real malware. It was shown that a high probability existed that a new malware instance was a variant of existing malware. Approximate matching was shown to detect more malware variants than exact matching, yet exact matching was shown to have comparable effectiveness. The exact matching classification system was found to perform efficiently in our evaluation with performance suitable for potential application in an Internet gateway or in desktop Antivirus.

References

[1] Symantec 2008, *Symantec internet security threat report: Volume xii*, Symantec.

[2] F-Secure 2007, *F-secure reports amount of malware grew by 100% during 2007*, viewed 19 August 2009, http://www.f-secure.com/en_EMEA/about-us/pressroom/news/2007/fs_news_20071204_1_eng.html

[3] Symantec 2009, *Symantec internet security threat report: Volume xiv*, Symantec.

[4] Heng, Y., Dawn, S., Manuel, E., Christopher, K. & Engin, K. 2007, *Panorama: Capturing system-wide information flow for malware detection and analysis*, Proceedings of the 14th ACM conference on Computer and communications security, Alexandria, Virginia, USA, ACM.

[5] Feily, M., Shahrestani, A. & Ramadass, S. 2009, *A survey of botnet and botnet detection*, Third International Conference on Emerging Security Information, Systems and Technologies (SECURWARE '09), pp. 268-273.

[6] Kolbitsch, C., Comparetti, P. M., Kruegel, C., Kirda, E., Zhou, X., Wang, X. F. & Santa Barbara, U. C. 2009, *Effective and efficient malware detection at the end host*, 18th USENIX Security Symposium.

[7] Griffin, K., Schneider, S., Hu, X. & Chiueh, T. 2009, *Automatic generation of string signatures for malware detection*, Recent Advances in Intrusion Detection: 12th International Symposium, RAID 2009, Saint-Malo, France, Springer.

[8] Kephart, J. O. & Arnold, W. C. 1994, *Automatic extraction of computer virus signatures*, 4th Virus Bulletin International Conference, pp. 178-184.

[9] Aho, A. V. & Corasick, M. J. 1975, 'Efficient string matching: An aid to bibliographic search', *Communications of the ACM*, vol. 18, pp. 340.

[10] Christodorescu, M., Kinder, J., Jha, S., Katzenbeisser, S. & Veith, H. 2005, *Malware normalization*, Technical Report #1539, University of Wisconsin, Madison, Wisconsin, USA.

[11] Mihai, C. & Somesh, J. 2004, *Testing malware detectors*, Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, Boston, Massachusetts, USA, ACM.

[12] Cullen, L. & Saumya, D. 2003, *Obfuscation of executable code to improve resistance to static disassembly*, Proceedings of the 10th ACM conference on Computer and communications security, Washington D.C., USA, ACM.

[13] Royal, P., Halpin, M., Dagon, D., Edmonds, R. & Lee, W. 2006, *Polyunpack: Automating the hidden-code extraction of unpack-executing malware*, Computer Security Applications Conference, pp. 289-300.

[14] Sharif, M., Lanzi, A., Giffin, J. & Lee, W. 2009, *Rotalume: A tool for automatic reverse engineering of malware emulators*.

[15] Panda Research 2007, *Mal(ware)formation statistics - panda research blog*, viewed 19 August 2009, http://research.pandasecurity.com/archive/Mal_2800_ware_2900_formation-statistics.aspx

[16] Stepan, A. 2006, *Improving proactive detection of packed malware*, Virus Bulletin Conference.

[17] Oberheide, J., Bailey, M. & Jahanian, F. 2009, *Polypack*, USENIX Workshop on Offensive Technologies (WOOT '09), Montreal, Canada.

[18] Graf, T. 2005, *Generic unpacking: How to handle modified or unknown pe compression engines*, Virus Bulletin Conference.

[19] 2010, *Upx: The ultimate packer for executables*, viewed 6 April 2010, http://upx.sourceforge.net/

[20] 2010, *Themida*, viewed 6 April 2010, http://www.themida.com/

[21] Kang, M. G., Poosankam, P. & Yin, H. 2007, *Renovo: A hidden code extractor for packed executables*, Workshop on Recurring Malcode, pp. 46-53.

[22] Boehne, L. 2008, Pandora's bochs: Automatic unpacking of malware, Thesis, University of Mannheim.

[23] Guizani, W., Marion, J. Y. & Reynaud-Plantey, D. 2009, *Server-side dynamic code analysis*, Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on, pp. 55-62.

[24] Kolter, J. Z. & Maloof, M. A. 2004, *Learning to detect malicious executables in the wild*, International Conference on Knowledge Discovery and Data Mining, pp. 470-478.

[25] Bilar, D. 2007, 'Opcodes as predictor for malware', *International Journal of Electronic Security and Digital Forensics*, vol. 1, pp. 156-168.

[26] Gheorghescu, M. 2005, *An automated virus classification system*, Virus Bulletin Conference, pp. 294-300.

[27] Aho, A. V., Sethi, R. & Ullman, J. D. 1986, *Compilers: Principles, techniques, and tools*, Addison-Wesley, Reading, MA.

[28] Kruegel, C., Kirda, E., Mutz, D., Robertson, W. & Vigna, G. 2006, 'Polymorphic worm detection using structural information of executables', *Lecture notes in computer science*, vol. 3858, pp. 207.

[29] Carrera, E. & Erdélyi, G. 2004, *Digital genome mapping–advanced binary malware analysis*, Virus Bulletin Conference, pp. 187-197.

[30] Ye, Y., Wang, D., Li, T. & Ye, D. 2007, *Imds: Intelligent malware detection system*, Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM.

[31] Christodorescu, M., Jha, S., Seshia, S. A., Song, D. & Bryant, R. E. 2005, *Semantics-aware malware detection*, Proceedings of the 2005 IEEE Symposium on Security and Privacy (S&P 2005), Oakland, California, USA.

[32] Horspool, R. N. & Marovac, N. 1979, 'An approach to the problem of detranslation of computer programs', *The Computer Journal*, vol. 23, pp. 223-229.

[33] Salton, G. & Mcgill, M. J. 1983, *Introduction to modern information retrieval*, McGraw-Hill New York.

[34] Kruegel, C., Robertson, W., Valeur, F. & Vigna, G. 2004, *Static disassembly of obfuscated binaries*, USENIX Security Symposium, vol. 13, pp. 18-18.

[35] Johannes, K., Florian, Z. & Helmut, V. 2009, *An abstract interpretation-based framework for control flow reconstruction from binaries*, Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation, Savannah, GA, Springer-Verlag.

[36] Daniel, K., Stner & Stephan, W. 2002, 'Generic control flow reconstruction from assembly code', *SIGPLAN Not.*, vol. 37, pp. 46-55.

[37] Theiling, H. 2000, *Extracting safe and precise control flow from binaries*, Proceedings of the Seventh International Conference on Real-Time Systems and Applications, IEEE Computer Society.

[38] Dalla Preda, M., Madou, M., De Bosschere, K. & Giacobazzi, R. 'Opaque predicates detection by abstract interpretation', *Algebraic Methodology and Software Technology*, pp. 81–95.

[39] Balakrishnan, G., Reps, T., Melski, D. & Teitelbaum, T. 2007, 'Wysinwyx: What you see is not what you execute', *Verified Software: Theories, Tools, Experiments*, pp. 202-213.

[40] Leder, F., Steinbock, B. & Martini, P. 2009, *Classification and detection of metamorphic malware using value set analysis*, Proc. of 4th International Conference on Malicious and Unwanted Software (Malware 2009), Montreal, Canada.

[41] Debray, K. C. S. & Townsend, T. K. G. 2009, *Automatic static unpacking of malware binaries*, Working Conference on Reverse Engineering - WCRE.

[42] Moser, A., Kruegel, C. & Kirda, E. 2007, *Limits of static analysis for malware detection*, Annual Computer Security Applications Conference (ACSAC).

[43] Lyda, R. & Hamrock, J. 2007, 'Using entropy analysis to find encrypted and packed malware', *IEEE Security and Privacy*, vol. 5, pp. 40.

[44] Josse, S. 2007, 'Secure and advanced unpacking using computer emulation', *Journal in Computer Virology*, vol. 3, pp. 221-236.

[45] Bellard, F. 2005, *Qemu, a fast and portable dynamic translator*, USENIX Annual Technical Conference, pp. 41–46.

[46] Raffetseder, T., Kruegel, C. & Kirda, E. 2007, 'Detecting system emulators', *Lecture notes in computer science*, vol. 4779, pp. 1.

[47] Min Gyung, K., Heng, Y., Steve, H., Stephen, M. & Dawn, S. 2009, *Emulating emulation-resistant malware*, Proceedings of the 1st ACM workshop on Virtual machine security, Chicago, Illinois, USA, ACM.

[48] Quist, D. & Valsmith 2007, *Covert debugging circumventing software armoring techniques*, Black Hat Briefings USA.

[49] Luk, C. K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J. & Hazelwood, K. 2005, *Pin: Building customized program analysis tools with dynamic instrumentation*, Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, ACM New York, NY, USA.

[50] Martignoni, L., Christodorescu, M. & Jha, S. 2007, *Omniunpack: Fast, generic, and safe unpacking of malware*, Proceedings of the Annual Computer Security Applications Conference (ACSAC), pp. 431-441.

[51] Dinaburg, A., Royal, P., Sharif, M. & Lee, W. 2008, *Ether: Malware analysis via hardware virtualization extensions*, Proceedings of the 15th ACM conference on Computer and communications security, ACM New York, NY, USA, pp. 51-62.

[52] Wu, Y., Chiueh, T. & Zhao, C. 2009, *Efficient and automatic instrumentation for packed binaries*, International Conference and Workshops on Advances in Information Security and Assurance, pp. 307-316.

[53] Sun, L., Ebringer, T. & Boztas, S. 2008, *Hump-and-dump: Efficient generic unpacking using an ordered address execution histogram*, International Computer Anti-Virus Researchers Organization (CARO) Workshop.

[54] Peter, N. Y. 1993, *Data structures and algorithms for nearest neighbor search in general metric spaces*, Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms, Austin, Texas, United States, Society for Industrial and Applied Mathematics, pp. 311-321.

[55] Paolo, C., Marco, P. & Pavel, Z. 1997, *M-tree: An efficient access method for similarity search in metric spaces*, Proceedings of the 23rd International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc.

[56] Bonfante, G., Kaczmarek, M. & Marion, J. Y. 2008, *Morphological detection of malware*, International Conference on Malicious and Unwanted Software, IEEE, Alexendria VA, USA, pp. 1-8.

[57] Gerald, R. T. & Lori, A. F. 2007, 'Polymorphic malware detection and identification via context-free grammar homomorphism', *Bell Labs Technical Journal*, vol. 12, pp. 139-147.

[58] Dullien, T. & Rolles, R. 2005, *Graph-based comparison of executable objects (english version)*, SSTIC.

[59] Hu, X., Chiueh, T. & Shin, K. G. *Large-scale malware indexing using function-call graphs*, Computer and Communications Security, Chicago, Illinois, USA, ACM, pp. 611-620.

[60] Babar, K., Khalid, F. & Pakistan, P. 2009, *Generic unpacking techniques*, International Conference On Computer, Control and Communication.

[61] Martignoni, L., Paleari, R., Roglia, G. F. & Bruschi, D. 2009, *Testing cpu emulators*, Proceedings of the eighteenth international symposium on Software testing and analysis, Chicago, IL, USA, ACM, pp. 261-272.

[62] Cifuentes, C. 1994, Reverse compilation techniques, Thesis, Queensland University of Technology.

[63] Moretti, E., Chanteperdrix, G. & Osorio, A. 2001, *New algorithms for control-flow graph structuring*, Software Maintenance and Reengineering, pp. 184.

[64] Wei, T., Mao, J., Zou, W. & Chen, Y. 2007, *Structuring 2-way branches in binary executables*, International Computer Software and Applications Conference, vol. 01, pp. 115-118.

[65] Baeza-Yates, R. & Navarro, G. 1998, *Fast approximate string matching in a dictionary*, South American Symposium on String Processing and Information Retrieval (SPIR'98), pp. 14-22.

[66] 2009, *Offensive computing*, viewed 21 September 2009, http://www.offensivecomputing.net

[67] 2009, *Mwcollect alliance*, viewed 21 September 2009, http://alliance.mwcollect.org