Classification of Malware Using Structured Control Flow

Silvio Cesare and Yang Xiang
School of Management and Information Systems
Centre for Intelligent and Networked Systems
Central Queensland University
Rockhampton, Queensland 4702, Australia

silvio.cesare@gmail.com; y.xiang@cqu.edu.au

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 flowgraph based malware classification. Static analysis is widely used for the classification but can be ineffective if malware undergoes a code packing transformation to hide its real content. This paper proposes a novel algorithm for constructing a control flow graph signature using the decompilation technique of structuring. Similarity between structured graphs can be quickly determined using string edit distances. To reverse the code packing transformation, a fast application level emulator is proposed. To demonstrate the effectiveness of the automated unpacking and flowgraph based classification, we implement a complete system and evaluate it using synthetic and real malware. The evaluation shows our system is highly effective in terms of accuracy in revealing all the hidden code, execution time for unpacking, and accuracy in classification.

Keywords: Network security, malware, structured control flow, unpacking.

1 Introduction

Malware, short for malicious software, means a variety of forms of hostile, intrusive, or annoying software or program code. Malware is a pervasive problem in distributed computer and network systems. Detection of malware is important to a secure distributed computing environment. The predominant technique used in commercial anti-malware systems to detect an instance of malware is through the use of malware signatures. Malware signatures attempt to capture invariant characteristics or patterns in the malware that uniquely identifies it. The patterns used to construct a signature have traditionally derived from the malware's machine code and raw file contents.

Traditional malware signatures ineffectively capture the invariant characteristics common in self-mutating and modified variants in a strain of malware. Static analysis provides alternative characteristics that can be used.

Static analysis incorporating ngrams (Kolter and Maloof 2004, Karim et al. 2005), edit distances (Gheorghescu 2005), and control flow (Carrera and Erdélyi 2004, Dullien and Rolles 2005, Briones and Gomez 2008) have been proposed. A malware's control flow information provides static analysis a characteristic that is identifiable across strains of malware variants. This characteristic is shared because variants of malware often reuse code from earlier strains and versions. This reuse of code can be identified through isomorphic and similar flow graphs.

To hinder static analysis, the malware's real content is frequently hidden using a code transformation known as packing. Packing is not solely used by malware. Packing is also used in software protection schemes and file compression for legitimate software, yet the majority of malware also uses the code packing transformation. In one month during 2007, 79% of identified malware was packed (Panda Research 2007). Additionally, almost 50% of new malware in 2006 were repacked versions of existing malware (Stepan 2006).

Unpacking is a necessary component to perform static analysis and reveal the hidden characteristics of malware. In the problem scope of unpacking, it can be seen that many instances of malware utilise identical or similar packers. Many of these packers are also public, and malware often employs the use of these public packers. Many instances of malware also employ modified versions of public packers. Being able to automatically unpack malware in any of these scenarios, in addition to unpacking novel samples, provides benefit for static analysis to occur.

Automated unpacking relies on typical behaviour seen in the majority of packed malware – hidden code is dynamically generated and then executed. The hidden code is naturally revealed in the process image during normal execution. Monitoring execution for the dynamic generation and execution of the malware's hidden code can be achieved through emulation. Emulation provides a safe and isolated environment for malware analysis.

In this paper we present a system that employs dynamic and static analysis to automatically unpack and classify a malware instance as a variant, based on similarities of control flow graphs.

This paper makes the following contributions. First, we propose a novel algorithm for approximate identification of flowgraphs based on treating decompiled structured flowgraphs as signatures. These signatures can then be used to query a malware database for approximate matches, using the edit distance to indicate similarity. Second, we propose and evaluate automated

unpacking using application level emulation that is fast enough for potential integration into desktop Antivirus. The automated unpacker is capable of unpacking a known samples and also capable of unpacking unknown samples. We also propose a novel algorithm for determining when to stop emulation during unpacking using entropy analysis. Finally, we implement and evaluate our ideas in a prototype system that performs automated unpacking and malware classification.

The structure of this paper is as follows. Section 2 describes related work in automated unpacking and malware classification. Section 3 refines the problem definition and our approach to the proposed complete classification system. Section 4 describes the design and implementation of our prototype system. Section 5 evaluates our prototype using synthetic and real malware samples. Finally, Section 6 summarizes and concludes the paper.

2 Related Work

Automated unpacking employing whole system emulation was proposed in Renovo (Kang et al. 2007) and Pandora's Bochs (Boehne 2008). Whole system emulation has been demonstrated to provide effective results against unknown malware samples, yet is not completely resistant to novel attacks. Renovo and Pandora's Bochs both detect execution of dynamically generated code to determine when unpacking is complete and the hidden code is revealed. An alternative algorithm for detecting when unpacking is complete was proposed using execution histograms in Hump-and-dump (Sun et al. 2008) . The Hump-and-dump was proposed as potentially desirable for integration into an emulator. Polyunpack (Royal et al. 2006) proposed a combination of static and dynamic analysis to dynamically detect code at runtime which cannot be identified during an initial static analysis. The main distinction separating our work from previously proposed automated unpackers is our use of application level emulation and an aggressive strategy to determine that unpacking is complete. The advantage of application level emulation over whole system emulation is significantly greater performance. Application level emulation for automated unpacking has had commercial interest (Graf 2005) but has realized few academic publications evaluating its effectiveness and performance.

Dynamic Binary Instrumentation was proposed as an alternative to using an instrumented emulator (Quist and Valsmith 2007) employed by Renovo and Pandora's Bochs. Omnipack (Martignoni et al. 2007) and Saffron (Quist and Valsmith 2007) proposed automated unpacking using native execution and hardware based memory protection features. This results in high performance in comparison to emulation based unpacking. The disadvantage of these approaches is in the use of the unpacking system on E-Mail gateways, which forces the provision of a virtual or emulated sandbox in which to run. A virtual machine approach to unpacking using x86 hardware extensions was proposed in Ether (Dinaburg et al. 2008). The use of such a virtual machine and equally to whole system emulator is the requirement to install a license for each guest operating system. This restricts desktop adoption which typically has a single license. Virtual machines are also inhibited by slow start-up times, which again are problematic for desktop use. The use of a virtual machine also prevents the system being cross platform as the guest and host CPUs must be the same.

Malware classification has been proposed using a variety of techniques (Kolter and Maloof 2004, Karim et al. 2005, Perdisci et al. 2008). A variation of ngrams, coined nperms has been proposed (Karim et al. 2005) to describe malware characteristics and subsequently used in a classifier. An alternative approach is using basicblocks of unpacked malware, classified using edit distances, inverted indexes and bloom filters (Gheorghescu 2005). The main disadvantage of these approaches is that minor changes to the malware source code can result in significant changes to the resulting bytestream after compilation. This change can significantly impact the classification. Flowgraph based classification is an alternative method that attempts to solve this issue by looking at control flow as a more invariant characteristic between malware variants.

The commercial automated unpacking and structural classification system Vxclass (Zynamics) is most related to our research. Vxclass presents a system for unpacking and malware classification based on similarity of flowgraphs. The algorithm in Vxclass is based on approximately matching flowgraphs by identifying fixed points in the graphs and successively matching neighbouring nodes (Carrera and Erdélyi 2004, Dullien and Rolles 2005, Briones and Gomez 2008). BinHunt (Gao et al. 2008) provides a more thorough test of flowgraph similarity by soundly identifying the maximum common subgraph, but at reduced levels of performance and without application to malware classification. Identifying common subgraphs of fixed sizes can also indicate similarity and has better performance (Kruegel et al. 2006).

Our research differs from previous flowgraph classification research by using a novel approximate control flow graph matching algorithm. Except for Krueger et al., the classification systems in previous research measure similarity in the callgraph and control flow graphs, where as our work relies entirely on the control flow graphs. Also distinguishing our work is the proposed automated unpacking system, which is integrated into the flowgraph based classification system.

3 Problem Definition and Our Approach

The problem of malware classification and variant detection is defined in this section. Additionally, an overview of our approach to design and implement a malware classification system is presented.

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 against each malware in the database. The similarity is measured as a real number between 0 and 1 - 0 indicating not at all similar, 1 indicating an identical 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 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 using the novel application of structuring. 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. Using the edit distance for approximate matches between each flowgraph, a similarity between flowgraphs can be constructed. The similarity of each control flowgraph in the program is accumulated to construct the final measure of program similarity and variant identification.

4 System Design and Implementation

In this section, the design and implementation of the malware classification system prototype is examined.

4.1 Identifying Packed Binaries Using Entropy Analysis

The malware classification system performs an initial analysis on the input binary to determine if it has undergone a code packing transformation. Entropy analysis (Lyda and Hamrock 2007), is used to identify packed binaries. The entropy of a block of data describes the amount of information it contains. It is calculated as follows:

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

where *p(i)* is the probability of the i^{th} unit of information in event *x*'s sequence of N symbols. For malware packing analysis, the unit of information is a byte value, N is 256, and an event is a block of data from the malware. 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 (Wu et al. 2009) 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 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

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

x86 Instruction Set Architecture (ISA): Much of the 32-bit x86 ISA has been implemented in the prototype. Extensions to the ISA, including SSE and MMX instructions, have been partially implemented. A partial implementation is adequate for the prototype as 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.

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 our prototype 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.

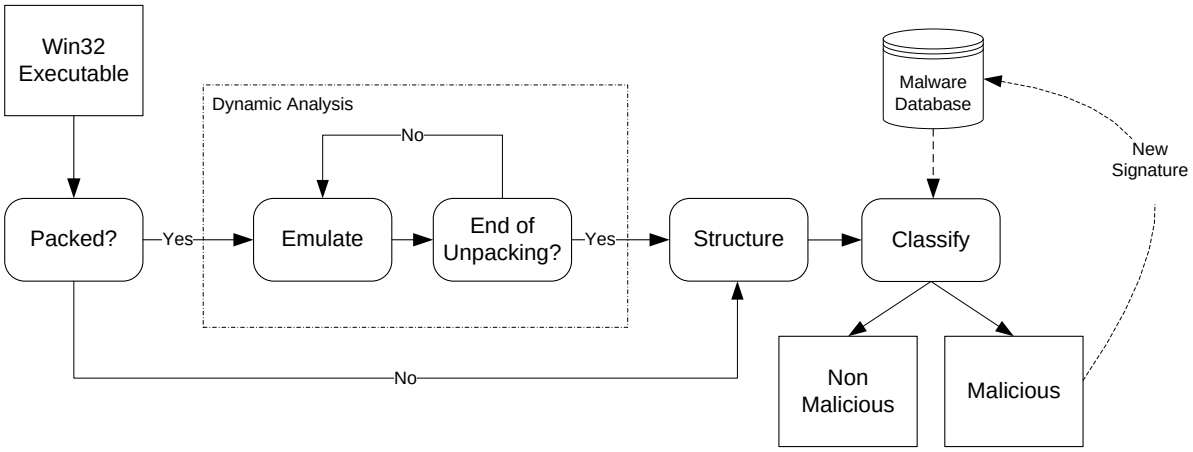

Figure 1: Block diagram of the malware classification system.

Windows API: The Windows API is the official system call interface provided by Windows. Our prototype system 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.

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 licensing issues should be avoided. Our implementation performs full dynamic library loading using the native libraries. This is done to provide a more faithful simulation.

Thread and Process Management: Multithreading in applications must be emulated. The prototype 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.

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 to the prototype as follows. We also make additional improvements to enable a mechanism to address anti-emulation code used by malware.

Instruction predecoding (Sharif et al.) 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.

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 (Bellard 2005).

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 (Babar et al. 2009). 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 to 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.

The prototype 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 (Martignoni et al. 2009). 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. The prototype debugger was modified to rewrite 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 (Kang et al. 2007). 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 executed will 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.

The malware classification prototype 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. The prototype 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 Flowgraph Based Signature Generation

The static analysis component of the malware classification system 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 from structuring its control flow into compact representation that is amenable to string matching.

To initiate the static analysis process, the memory image of the binary is disassembled using speculative disassembly (Kruegel et al. 2004). 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 callsite belongs to an existing procedure. Once disassembled the disassembly is translated into an intermediate representation. Using an intermediate representation is not strictly necessary; however the malware classification prototype is built on a more general binary analysis platform which uses the intermediate form. The intermediate representation is used to generate a control flow graph for each identified procedure. The control flow graph is then structured into a signature represented as a character string. The signature is also associated with a weight relative to all sum of all weighted signatures in the program. The weight of procedure x is formally defined as:

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

where s_i is signature of procedure i in the binary.

4.4.1 Signatures using Structured Control Flow for Approximate Matches

Malware classification using approximate matches of signatures can be performed, and intuitively, using approximate matches of a control flow graph should enable identification a greater number of malware variants. In our approach we use structuring. 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 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 the prototype is a modified algorithm of that proposed in the DCC decompiler (Cifuentes 1994). 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 improvements to this algorithm

to reduce the generation of unstructured branches have been proposed (Moretti et al. 2001, Wei et al. 2007), 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 3.

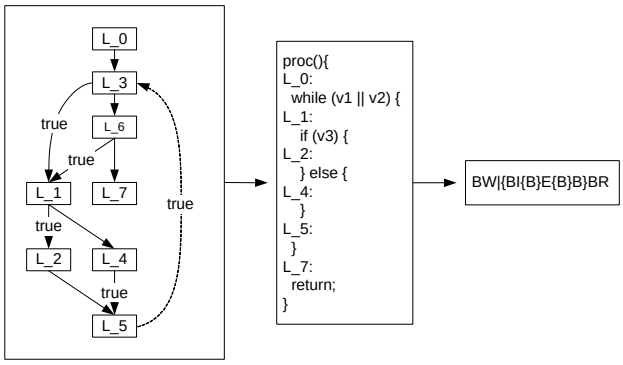

Figure 2: The relationship between a control flow graph, a high level structured graph, and a signature.

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 3: The grammar to represent a structured control flow graph signature.

4.5 Malware Classification

To classify an input binary, the analysis makes use of a malware database. The database contains the signatures, represented as structured graphs, of known malware. For each procedure in the input binary, the database is queried using an approximate dictionary search. The results are accumulated to give a measure of similarity between the input binary and malware in the database.

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.

To perform the approximate dictionary search, any signature within an allowable number of errors is identified as a positive match. The edit distance between signatures gives the number of errors, and the search is performed using BK Trees (Baeza-Yates and Navarro 1998). The search algorithm is faster than an exhaustive comparison of each signature in the dictionary. In our prototype, we maintain a separate database for each malware instance and perform classification against all instances one at a time. This has resulted in greater performance than using a global database of all malware.

The similarity ratio (Gheorghescu 2005) 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 a 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. Our prototype 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. Using the similarity ratio s as a threshold, the number of allowable errors in signature x is defined as $len(x)(1-s)$.

For a particular malware, once a matching graph is found, this graph is ignored for subsequent searches of the remaining graphs in the input binary. If a graph has multiple matches in a particular malware and it is uncertain which procedure should be selected as a match, the greedy solution is taken. The graph that is weighted the most is selected. Two weights are possible for each procedure from the input binary during dictionary queries, as the procedure may be considered belonging to either the input binary, or the malware binary in the database.

For each malware that has matching signatures, the similarity ratios of those signatures are accumulated proportional to their weights. As two weights are possible, this results in calculating two asymmetric similarities: a similarity that identifies how much of the input binary is approximately found in the database malware, and a similarity to show how much of the database malware is approximately found in the input binary. 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_i} is the similarity ratio between the i^{th} control flow graph of

the input binary and the matching graph in the malware database, and $weight_{x_i}$ is the weight of the cfg where *x* is either the input binary or the malware binary in the database.

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.

Program similarity is not symmetric and therefore *S(i,d)* and *S(d,i)* are not guaranteed the same result, although in practice they are generally identical. This difference is due to the use of heuristics, as a control flow graph signature can potentially be found to have multiple matches when calculating the similarity between two binaries. The heuristic used is to select a single signature. A greedy selection is taken, based on the weights of the matching signatures. In *S(i,d)*, the input binary defines the weights used in the greedy selection. Alternatively, in *S(d,i)*, the database malware defines the weights which will be used. Because the weights and therefore greedily selected signatures can be different, the resulting program similarities may be different.

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 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.

4.6 Discussion

It is possible to generate a signature using a faster and simpler method than structuring (Carrera and Erdélyi 2004). This approach takes note that if the signatures of two graphs are not the same, then the graphs are not isomorphic. The converse is used to indicate matching graphs. To generate a signature, the algorithm orders the nodes in the control flow graph, e.g. using a depth first order. A signature subsequently consists of a list of graph edges using the ordered nodes as node labels. The potential advantage of this method is that classification using exact matches of signatures can be performed very efficiently.

Another potential design change is using an alternative method to establish the program similarity. It may be desirable to identify only that malware is approximately found as a subset in another binary, in which case a single asymmetric similarity may be used. This has use in virus detection.

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.

Malware classification has inherent problems also, and may fail to perform correctly. 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 Evaluation

In this section we describe experiments that were used to evaluate automated unpacking and flowgraph based classification using our prototype.

5.1 Unpacking Synthetic Samples

OEP Detection: To verify our system correctly performs hidden code extraction, we tested the 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 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. 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 the better. The higher the result in column 5 the better.

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

hostname.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

calc.exe

Table 1: Metrics on identifying the original entry point in packed samples.

The results show that unpacking most of the samples reveals some or most of the hidden code as expected. 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 classification system in the static analysis phase.

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 32bit Windows Vista Home Premium with Service Pack 1. The performance of the unpacking system, shown in table 2. The running time is total time minus start-up time of 0.60s. Binaries that failed to pack correctly are marked as fail.

Name	Time (s)	Num. Instr.
mew	0.13	56042
fsg	0.13	58138
upx	0.11	61654
packman	0.13	123959
npack	0.14	129021
aspack	0.15	161183
pe compact	0.14	179664
expressor	0.20	620932
winupack	0.20	632056
yoda’s protector	0.15	659401
rlpack	0.18	916590
telock	0.20	1304163
acprotect	0.67	3347105
pespin	0.64	10482466

hostname.exe

Name	Time (s)	Num. Instr.
mew	1.21	12691633
fsg	0.23	964168
upx	0.19	1008720
packman	0.28	1999109
npack	0.40	2604589
aspack	0.51	4078540
pe compact	0.83	7691741
expressor	fail	fail
winupack	0.93	7889344
yoda’s protector	0.24	2620100
rlpack	0.56	7632460
telock	fail	fail
acprotect	0.53	5364283
pespin	1.60	27583453

calc.exe

Table 2: Running time to perform unpacking.

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

5.2 Flowgraph Based Malware Classification

The malware classification prototype was evaluated using real malware samples. The samples were chosen to mimic a selection of the malware and evaluation metrics in previous research (Carrera and Erdélyi 2004). Netsky, Klez, and Roron malware variants were obtained through public databases. A number of the malware samples were packed. The classification system automatically identifies and unpacks such malware as necessary. Table 3 evaluates the complete system. Highlighted cells identify a malware variant, defined as having a similarity equal or exceeding 0.60. A flowgraph is classed as being a variant of another flowgraph if the similarity ratio is equal or in excess of 0.9.

	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	

The Klez family of malware.

	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	

The Netsky family of malware.

Table 3: The similarity matrix for two malware families (e.g. Klez.a).

	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	

Similarity ratio threshold of 1.0.

	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	

Default similarity ratio threshold of 0.9.

Table 4: The similarity matrices for the Roron family of malware (e.g. Roron.ao).

	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 5: The similarity matrix for non similar malware and programs.

The results demonstrate that the system finds high similarities between samples. Table 4 shows the difference in the similarity matrix when the threshold for the similarity ratio is increased to 1.0. Differences between 5% and 30% were noted across the variety of malware variants using the two similarity ratio thresholds.

To evaluate the generation of false positives in the classification system table 5 shows classification among non similar binaries. Table 6 shows a more thorough evaluation of 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. There 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.

Similarity	Matches
0.0	105497
0.1	2268
0.2	637
0.3	342
0.4	199
0.5	121
0.6	44
0.7	72
0.8	24
0.9	20
1.0	6

Table 6: Histogram of similarities between executable files in Windows system directory.

6 Conclusion

Malware can be classified according to similarity in its flowgraphs. This analysis is made more challenging by packed malware. In this paper we proposed fast algorithms to unpack malware using application level emulation, and perform malware classification using the edit distance between structured control flow graphs. We implemented and evaluated a prototype. It was demonstrated that the automated unpacking system was fast enough for desktop integration. The automated unpacking was also demonstrated to work against a promising number of synthetic samples using known packing tools, with high speed. To detect the completion of unpacking, we proposed and evaluated the use of entropy analysis. It is shown that our system can successfully identify variants of malware.

7 References

Babar, K., Khalid, F. & Pakistan, P. (2009): Generic Unpacking Techniques. *International Conference On Computer, Control and Communication*.

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

Bellard, F. (2005): QEMU, a fast and portable dynamic translator. *USENIX Annual Technical Conference*, 41–46.

Boehne, L. (2008): Pandora's Bochs: Automatic Unpacking of Malware. University of Mannheim.

Briones, I. & Gomez, A. (2008): Graphs, Entropy and Grid Computing: Automatic Comparison of Malware. *Virus Bulletin Conference*, 1-12.

Carrera, E. & Erdélyi, G. (2004): Digital genome mapping–advanced binary malware analysis. *Virus Bulletin Conference*, 187-197.

Cifuentes, C. (1994): Reverse compilation techniques. Queensland University of Technology.

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*, 51-62, ACM New York, NY, USA.

Dullien, T. & Rolles, R. (2005): Graph-based comparison of Executable Objects (English Version). *SSTIC*.

Gao, D., Reiter, M. K. & Song, D. (2008): Binhunt: Automatically finding semantic differences in binary programs. *Information and Communications Security*, **5308**:238–255, Springer.

Gheorghescu, M. (2005): An automated virus classification system. *Virus Bulletin Conference*, 294-300.

Graf, T. (2005): Generic unpacking: How to handle modified or unknown PE compression engines. *Virus Bulletin Conference*.

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

Karim, M. E., Walenstein, A., Lakhotia, A. & Parida, L. (2005) Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, **1** (1)**:**13-23.

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

Kruegel, C., Kirda, E., Mutz, D., Robertson, W. & Vigna, G. (2006) Polymorphic worm detection using structural information of executables. *Lecture notes in computer science*, **3858:**207.

Kruegel, C., Robertson, W., Valeur, F. & Vigna, G. (2004): Static disassembly of obfuscated binaries. *USENIX Security Symposium*, **13**:18-18.

Lyda, R. & Hamrock, J. (2007) Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, **5** (2)**:**40.

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

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, 261-272, ACM.

Moretti, E., Chanteperdrix, G. & Osorio, A. (2001): New algorithms for control-flow graph structuring. *Software Maintenance and Reengineering*, 184.

Mal(ware)formation statistics - Panda Research Blog: Panda Research, http://research.pandasecurity.com/archive/Mal_2800_ware_2900_formation-statistics.aspx. 19 August 2009.

Perdisci, R., Lanzi, A. & Lee, W. (2008): McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables. *Proceedings of the 2008 Annual Computer Security Applications Conference*, 301-310, IEEE Computer Society Washington, DC, USA.

Quist, D. & Valsmith (2007): Covert Debugging Circumventing Software Armoring Techniques. *Black Hat Briefings USA*.

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*, 289-300.

Sharif, M., Lanzi, A., Giffin, J. & Lee, W. Rotalume: A Tool for Automatic Reverse Engineering of Malware Emulators. INC., I. G.

Stepan, A. (2006): Improving proactive detection of packed malware. *Virus Bulletin Conference*, **1**.

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*.

Wei, T., Mao, J., Zou, W. & Chen, Y. (2007): Structuring 2-way branches in binary executables. *International Computer Software and Applications Conference*, **01**: 115-118.

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*, 307-316.

VxClass: Zynamics, http://www.zynamics.com/vxclass.html.