# Malwise—An Effective and Efficient Classification System for Packed and Polymorphic Malware

Silvio Cesare, *Student Member*, *IEEE*, Yang Xiang, *Senior Member*, *IEEE*, and Wanlei Zhou, *Senior Member*, *IEEE*

**Abstract**—Signature-based malware detection systems have been a much used response to the pervasive problem of malware. Identification of malware variants is essential to a detection system and is made possible by identifying invariant characteristics in related samples. To classify the packed and polymorphic malware, this paper 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. An exact flowgraph matching algorithm is employed that uses string-based signatures, and is able to detect malware with near real-time performance. Additionally, a more effective approximate flowgraph matching algorithm is proposed that uses the decompilation technique of structuring to generate string-based signatures amenable to the string edit distance. We use real and synthetic malware to demonstrate the effectiveness and efficiency of Malwise. Using more than 15,000 real malware, collected from honeypots, the effectiveness is validated by showing that there is an 88 percent probability that new malware is detected as a variant of existing malware. The efficiency is demonstrated from a smaller sample set of malware where 86 percent of the samples can be classified in under 1.3 seconds.

**Index Terms**—Computer security, malware, control flow, structural classification, 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. 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]. 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 strings of the malware's machine code and raw file contents [3], [4]. String-based signatures have remained popular in commercial systems due to their high efficiency, but can be ineffective in detecting malware variants.

Malware variants often have distinct byte-level representations while in principal belong to the same family of malware. The byte-level content is different because small changes to the malware source code can result in significantly different compiled object code. In this paper, we describe malware variants with the umbrella term of polymorphism. Polymorphism describes related malware sharing a common history of code. Code sharing among variants can be derived from autonomously self-mutating malware, or manually copied by the malware creator to reuse previously authored code.

### 1.1 Existing Approaches and Motivation

Static analysis incorporating n-grams [5], [6], edit distances [7], API call sequences [8], and control flow [9], [10], [11] have been proposed to detect malware and their polymorphic variants. However, they are either ineffective or inefficient in classifying packed and polymorphic malware.

A malware's control flow information provides a characteristic that is identifiable across strains of malware variants. Approximate matchings of flowgraph-based characteristics can be used in order to identify a greater number of malware variants. Detection of variants is possible even when more significant changes to the malware source code are introduced.

Control flow has proven effective [9], [11], [12], and fast algorithms have been proposed to identify exact isomorphic whole program control flow graphs [13] and related information [14], yet approximate matching of program structure has shown to be expensive in runtime costs [15]. Poor performance in execution speed has resulted in the absence of approximate matching in endhost malware detection.

To hinder the static analysis necessary for control flow analysis, the malware's real content is frequently hidden

• *The authors are with the School of Information Technology, Deakin University, Melbourne Burwood Campus, 221 Burwood Highway, Burwood, VIC 3125, Australia.*
*E-mail: {s.cesare, yang, wanlei}@deakin.edu.au.*

*Manuscript received 19 May 2011; revised 18 Sept. 2011; accepted 21 Feb. 2012; published online 7 Mar. 2012.*
*Recommended for acceptance by M. Guo.*
*For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2011-05-0333.*
*Digital Object Identifier no. 10.1109/TC.2012.65.*

using a code transformation known as packing [16]. 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 percent of identified malware was packed [17]. Additionally, almost 50 percent of new malware in 2006 were repacked versions of existing malware [18].

Unpacking is a necessary component to perform static analysis and to reveal the hidden characteristics of malware. In the problem scope of unpacking, it can be seen that many instances of malware utilize 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 in revealing the malware's real content—a necessary component for static analysis and accurate classification.

Automated unpacking relies on typical behavior 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 [19]. Emulation provides a safe and isolated environment for malware analysis.

Malware detection has been investigated extensively, however shortcomings still exist. For modern malware classification approaches, a system must be developed that is not only effective against polymorphic and packed malware, but that is also efficient. Unless efficient systems are developed, commercial antivirus will be unable to implement the solutions developed by researchers. We believe combining effectiveness with real-time efficiency is an area of research which has been largely ignored. For example, the malware classification investigated in [5], [6], [9], [10], [11] has no analysis or evaluation of system efficiency. We address that issue with our implementation and evaluation of Malwise.

In this paper, we present an effective and efficient 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.

### 1.2 Contributions

This paper makes the following contributions. First, we propose using string-based signatures to represent control flow graphs and a fast classification algorithm based on set similarity to identify related programs in a database. Second, we propose using two algorithms to generate string signatures for exact and approximate identification of flowgraphs. Exact matching uses a graph invariant as a string to heuristically identify isomorphic flowgraphs. Approximate matching uses decompiled structured flowgraphs as string signatures. These signatures have the property that similar flowgraphs have similar strings. String similarity is based on the string edit distance. Third, we propose and evaluate automated unpacking using application-level emulation that is equally capable of desktop antivirus integration. The automated unpacker is capable of unpacking known samples and is also capable of unpacking unknown samples. We also propose an algorithm for determining when to stop emulation during unpacking using entropy analysis. Finally, we implement and evaluate our ideas in a novel prototype system called Malwise that performs automated unpacking and malware classification.

### 1.3 Structure of the Paper

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 malware classification system; Section 4 describes the design and implementation of our prototype Malwise system; Section 5 evaluates Malwise using real and synthetic malware samples; finally, Section 8 summarizes and concludes the paper.

## 2 Related Work

### 2.1 Automated Unpacking

Automated unpacking employing whole system emulation was proposed in Renovo [19] and Pandora's Bochs [20]. Whole system emulation has been demonstrated to provide effective results against unknown malware samples, yet is not completely resistant to novel attacks [21]. 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 [22]. The Hump and dump was proposed as potentially desirable for integration into an emulator. Polyunpack [16] 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 [23] but has realized few academic publications evaluating its effectiveness and performance.

Dynamic binary instrumentation was proposed as an alternative to using an instrumented emulator [24] employed by Renovo and Pandora's Bochs. Omnipack [25] and Saffron [24] 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 unpacking using native execution is evident on e-mail gateways because a virtual machine or emulator is required to execute the malware. A virtual machine approach to unpacking, using x86 hardware extensions, was proposed in Ether [26]. The use of such a virtual machine and equally to a 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.

### 2.2 Polymorphic Malware Classification

Malware classification has been proposed using a variety of techniques [5], [6], [27]. A variation of n-grams, coined n-perms has been proposed [6] to describe malware characteristics and subsequently used in a classifier. An alternative approach is using the basic blocks of unpacked malware, classified using edit distances, inverted indexes and bloom filters [7]. 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. Abstract interpretation has been used to construct tree structure signatures [28] and model malware semantics [29]. However, these approaches do not perform inexact matching. Windows API call sequences have been proposed [8] as an invariant characteristic, but correctly identifying the API calls can be problematic [20] when code packing obscures the result. 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.

Flowgraph-based classification utilizes the concept of approximating the graph edit distance, which allows the construction of similarity between graphs. The commercial system Vxclass [30] presents a system for unpacking and malware classification based on similarity of flowgraphs. The algorithm in Vxclass is based on identifying fixed points in the graphs and greedily matching neighboring nodes [9], [10], [11]. BinHunt [31] 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 [32]. SMIT [15] identifies malware with similar call graphs using minimum cost bipartite graph matching and the Hungarian algorithm, improving upon the greedy approach to graph matching investigated in earlier research. SMIT additionally uses metric trees to improve performance on graph-based database searches. Tree automata to quickly recognize whole program control flow graphs has also been proposed [13] but this approach only identifies isomorphisms or subgraph ismorphisms, and not approximate similarity. Context free grammars have been extracted from malware to describe control flow [14], but this research only identifies whole program equivalence. Clustering is related to classification and has been proposed for use on call graphs using the graph edit distance to construct similarity matrices between samples [33].

### 2.3 The Difference between Malwise and Previous Work

Our research differs from previous flowgraph classification research by using a novel approximate control flow graph matching algorithm employing structuring. We are the first to use the approach of structuring and decompilation to generate malware signatures. This allows us to use string-based techniques to tackle otherwise infeasible graph problems. We use an exact matching algorithm that performs in near real time while still being able to identify approximate matches at a whole program level. The novel set similarity search we perform enables the real-time classification of malware from a large database. No prior related research has performed in real time. The classification systems in most of the previous research measure similarity in the call graph and control flow graphs, whereas our work relies entirely on the control flow graphs at a procedure level. Additionally, 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. The problem summary is to use instance-based learning and perform a similarity search over a malware database. Additionally defined in this section is an overview of our approach to design the Malwise system.

### 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 nonmalicious. 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 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.

Two approaches are employed to generate and compare flowgraph signatures. The system design is presented in Fig. 1.

Two flowgraph matching methods are used to achieve the goal of either effectiveness or efficiency. A brief introduction is provided here. More details will be given in Sections 4.5 and 4.6.

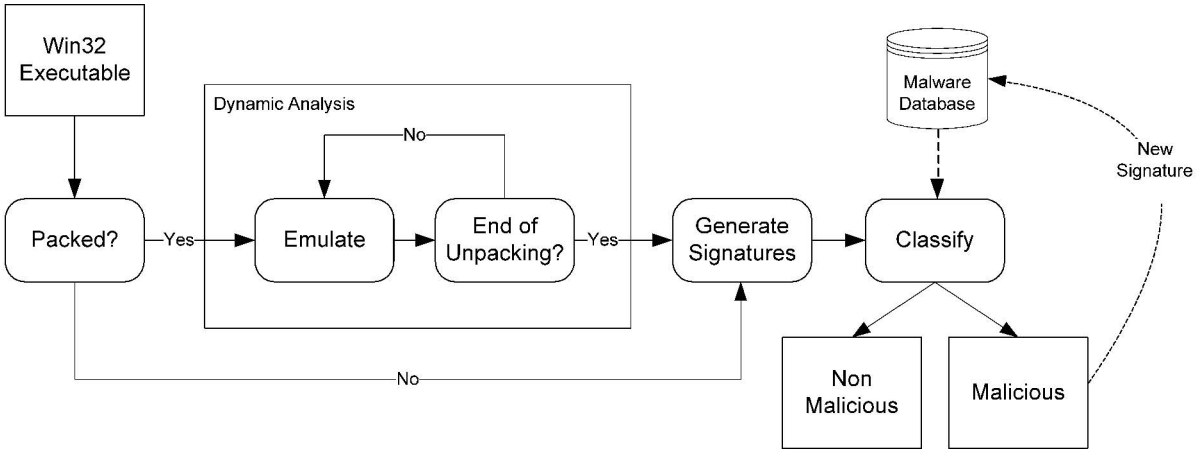



Fig. 1. Block diagram of the malware classification system.

*Exact 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.

*Approximate 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 edit distance.

## 4 System Design and Implementation

### 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 [34] 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:

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

where *p(i)* is the probability of the $i$th unit of information in event $x$'s sequence of $N$ symbols. For 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 are typically characterized as being encrypted or compressed; therefore, high entropy in the malware can indicate packing.

An analysis most similar to Uncover [35] is employed. Identification of packed malware is established if there exist sequential blocks of high entropy data in the input binary. Further refinements to this approach are proposed in [36]. In our work, false positives only impact upon efficiency and not effectiveness.

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.

### 4.3 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 OEP in Renovo [19]. 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 OEP 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 exist high entropy packed data that have not been used by the packer during execution, then they remain to be unpacked. To determine if a particular stage of unpacking represents the OEP, 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 analyzed data is low, then it is presumed that no more compressed or encrypted data are 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 OEP 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 OEP at this location.

### 4.4 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 Sections 4.5 and 4.6.

To initiate the static analysis process, the memory image of the binary is disassembled using speculative disassembly [37]. 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. 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 on a more general binary analysis platform that uses 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 transformed into a signature represented as a character string. The signature is also associated with a weight, described in Sections 4.5 and 4.6. The weight intuitively represents the importance of the signature when used to determine program similarity.

### 4.5 Exact Flowgraph Matching

It is possible to generate a signature using a fast and simple method than structuring, if the matching algorithm only identifies isomorphisms [9]. 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 Fig. 2.

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.

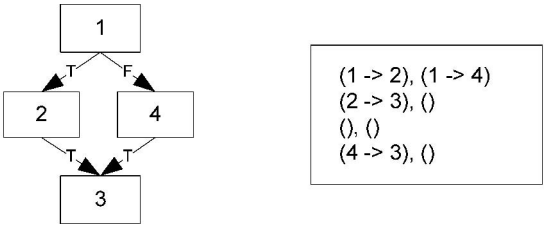



Fig. 2. A depth first-ordered flowgraph and its signature.

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 $O(\log(N))$.

### 4.6 Approximate Flowgraph Matching

Malware classification using approximate matches of signatures is performed, and intuitively, using approximate matches of a control flow graph, instead of exact isomorphisms, 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 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 [38]. 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 [39], [40]. 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.

```
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 '}'
```

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

The grammar for a resulting signature is defined in Fig. 3. Fig. 4 shows an example of the relationship between a control flow graph, a high-level structured graph, and a resulting signature.

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 [7] 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. 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.

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 [41]. 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.

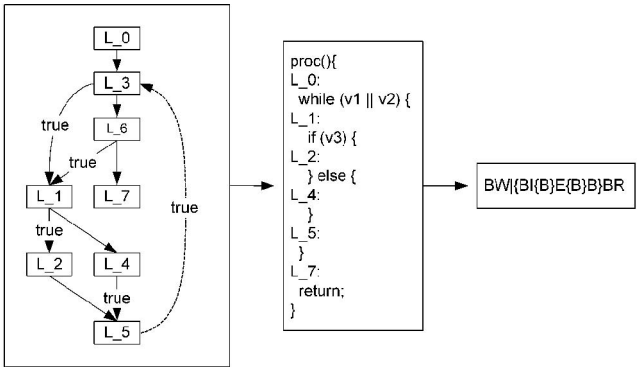



Fig. 4. The relationship between a control flow graph, a high-level structured graph, and a signature.

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.

### 4.7 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 an 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 of assignment is shown in Fig. 5.

Two weights are associated with each matching flowgraph signature as shown in example Fig. 6. 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

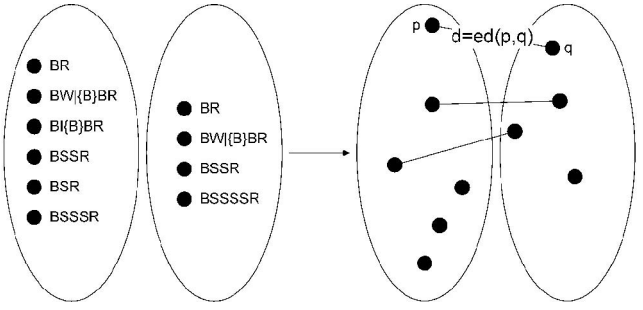



Fig. 5. Assignment of flowgraph strings between sets.

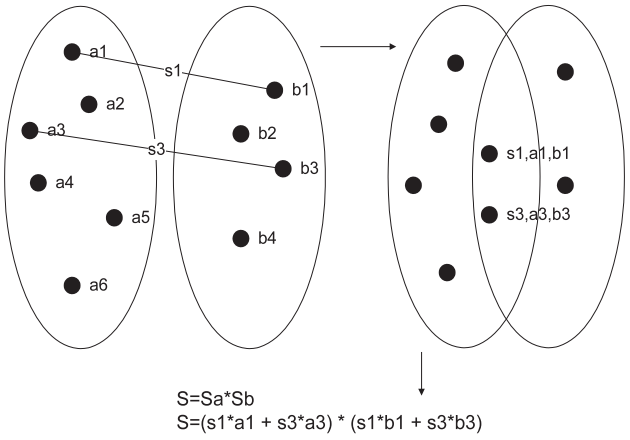



Fig. 6. Malware classification using set similarity.

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.

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 five 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.

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. Program similarity exceeding 0.95 is used in Malwise to define signatures excessively similar.

To perform approximate matching and classification in Malwise, we maintain a separate flowgraph database for each malware instance and perform classification against instances one at a time. This has resulted in greater performance than using a global database of all malware. Exact matching employs a global database.

### 4.8 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. To improve the performance, the similarity between programs represented as sets, can utilize 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.

Our 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 approach is that some flowgraph strings have many matching malware. To handle this problem, we divide the classification process into two stages. Pseudocode to describe the algorithm is given in Fig. 7. In the first stage, we only build the asymmetric similarity for flowgraphs which are associated with a unique or near unique malware (shown in section A of the pseudocode). The unique flow graphs are returned from the unique_cfg_matches function and the remaining duplicates are returned from the duplicate_cfg_matches function. Both functions use a global database of flow graphs whereas the matching_cfgs_in_specific_db use a database specific to a particular malware. At completion of processing uniquely matching malware, we prune those that cannot have a program similarity above 0.6 (shown in section B). Finally, we process the remaining flowgraph strings, but we do not employ the entire flowgraph database, but instead use a local database for each of the malware remaining from the previous stage (shown in section C). The seenBefore function implements a greedy matching of individual flow graph signatures (known here as malware signatures). Once a malware signature (control flow graph) has been processed, it cannot contribute any more to the result. Finally, after the algorithm is complete we 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.

### 4.9 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 $O(N\log(M))$ 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 $O(N\log(M) + AN\log(N))$, 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 [15] 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 percent of the database size $M$, was pruned when identifying the 10 nearest neighbors in a search utilizing metric trees [15]. Our algorithm has similar intentions andcomparable results in

```
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)
{
        // section A
        similarityTogo = 1.0
        foreach u in unique_cfg_matches(db, cfgs(in)) {
                solutions.reset()
                ProcessMatch(u, similarityTogo)
                similarityTogo -= weight_of_input_cfg(u)
        }
        // section B
        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)
        }
        // section C
        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
}
```

Fig. 7. Set similarity search.

identifying malware variants, and performs significantly more efficiently. The runtime complexity of a typical multipattern string matching algorithm used in antivirus systems, employing the Aho-Corasick algorithm [42] is linear to the size of the input program and number of identified matches. The disadvantage of this approach is that preprocessing 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 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 [13]. Our system is more resilient to false positives under these conditions because increased flowgraph complexity enables more precise signatures.

## 4.10 Discussion

### *4.10.1 Similarity Thresholds*

The threshold to determine if two programs are similar, in either exact flowgraph matching or approximate flowgraph matching, is empirically decided in Malwise. Likewise as is the similarity ratio between flow graphs. The actual figures are decided by investigating a huge number of real-life malware samples. This approach is currently adopted by most antivirus systems. It is a desirable feature that the malware classification system can adaptively select the thresholds. Machine learning-based approach can be taken to achieve this. As the main focus of this research is to develop an effective and efficient system to solve the polymorphic malware problem, we leave this as our future work.

#### 4.10.2 Failure of Unpacking

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 antiemulation code may be treated using custom handlers to perform the simulation where antiemulation code is detected.

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

Finally, it is possible that malware is packed yet has relatively low entropy. This is not a common scenario; however, if more malware employs this mechanism then n-gram analysis could be used to detect packing [27] or alternatively, unpacking be applied to all binaries.

#### 4.10.3 Effectiveness of Static Analysis

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 at a postprocessing 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 the experiments to evaluate automated unpacking and flowgraph-based classification in Malwise.

TABLE 1
Identifying the OEP in Packed Samples

| 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*

### 5.1 Unpacking Synthetic Samples

#### 5.1.1 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 7,680 bytes in size, and calc.exe is 114,688 bytes.

The OEP 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. 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

TABLE 2
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 |

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 results in columns 3 and 4 indicate better performance. The closer result of 100 percent in column 5 indicates better performance. A score of 100 percent indicates a perfect result in unpacking.

The results show that unpacking the samples reveals most of the hidden. 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 are 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. In total, 43 percent 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.

#### 5.1.2 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 2. The running time is total time minus start-up time of 0.60 s. Binaries that failed to pack correctly are marked as fail. The number of instructions emulated during unpacking is also shown.

TABLE 3
Malware Similarity 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).*

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

### 5.2 Malware Classification

#### 5.2.1 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 [9]. The malware was obtained through a public database [43]. 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.

Tables 3 and 4 evaluate 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. Assuming antivirus vendors have correctly identified malware variants as belonging to the correct families, our evaluation should identify members of these families as being variants of each other. The more variants our system detects, the more effective it is. Highlighted cells

TABLE 4
Malware Similarity 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 | c | 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).*

identify a malware variant, defined as having a similarity equal or exceeding 0.60. In approximate matching, a flowgraph is classed as being a variant of another flowgraph if the similarity ratio is equal or in excess of 0.9. To improve the performance of exact matching, procedures with less than five 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. This affects the individual flow graph comparisons, but the similarity threshold between the set of flow graphs is still 0.6. Differences of up to 30 percent 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.

### 5.2.2 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 [44] network. The malware samples were sorted according to collection time, and processed in order. In total, 94.4 percent of malware samples were found to have a similarity of more than 95 percent to previously classified malware in the set. A total of 863 representative malware signatures were stored in the database, where none were more than 95 percent similar to other signatures. It was found that 88.26 percent of malware were detected as variants of previously classified

TABLE 5
Malware Similarity Using Approximate Matching 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 | |

*Roron*

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 percent of malware were 100 percent similar to existing malware, once unpacked. This corroborates research [18] 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 5.2.3 to validate the results.

### 5.2.3 Efficiency of Exact Matching

A total of 809 malware samples with unique MD5 hashes were collected between 29-04-2009 and 17-05-2009 from honeypots in the mwcollect alliance network [44] 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. A total of 754 samples were found to have at least one other sample in the set which was a variant. Fig. 8 evaluates the speed of processing these malware samples, including unpacking and classification time but excluding the loading time of the malware database. We present the data noncumulatively so that we can identify groupings of expected processing times. For example, the most common processing time of malware is a little over 1 second followed next by taking less than half a second for processing. 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 used in the unpacking performance testing. In total, 86 percent of the

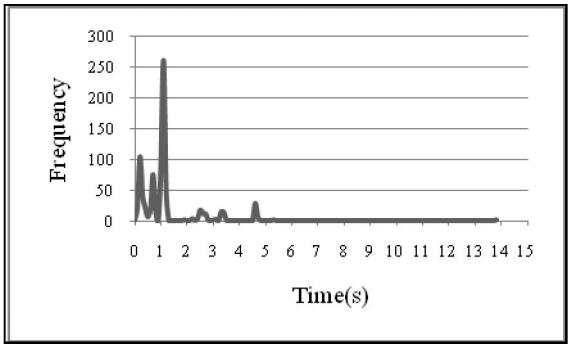



Fig. 8. Malware processing time.

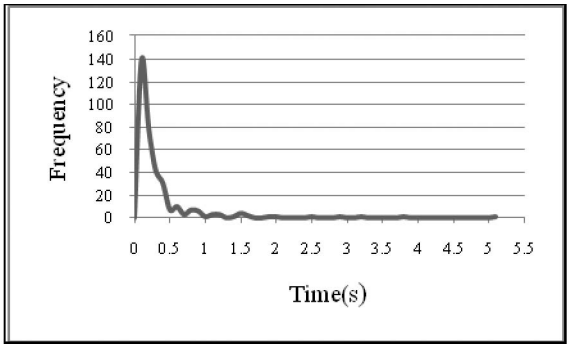



Fig. 9. Benign processing time.

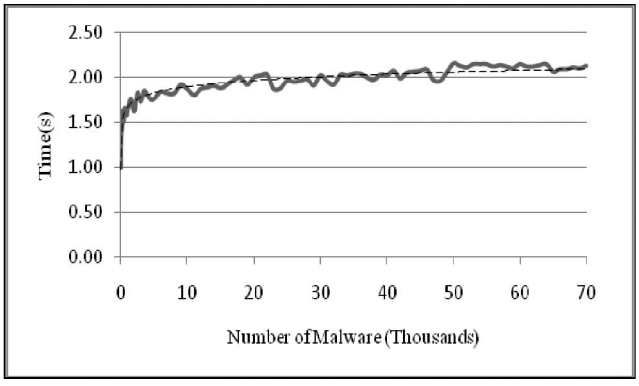



Fig. 10. Scalability of classification (100,000 repetitions).

malware were processed in under 1.3 seconds. The only malware that was not processed in under 5 seconds instead took nearly 14 seconds. This was because nearly 163 Million instructions were emulated during unpacking. This is possibly the result of an antiemulation 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 Fig. 9. The median time to perform classification was 0.25 seconds. The slowest sample classified required 5.12 seconds. Only six 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 end host adoption.

#### 5.2.4 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 percent 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 Fig. 10. 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 6
Nonsimilar Program Similarity 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 | |

TABLE 7
Nonsimilar Program Similarity 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 8
Histogram of Similarities between Benign Files

| 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 |

#### 5.2.5 Malwise's Resilience to False Positives

To evaluate the generation of false positives in Malwise, Tables 6 and 7 show classification among nonsimilar binaries using approximate and exact matching.

To further evaluate the exact matching algorithm against false positives, the malware database created from the 809 samples in Section 5.2.3 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.

To continue evaluation of exact and approximate matching, Table 8 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 five basic blocks, which resulted in some binaries being ineligible for analysis.

Although our system has a limited number of false positives, some discrepancies still exist. We believe that the size of malicious and benign binaries play a part in this. For small binaries with small call graphs and few procedures, the resulting signatures are accordingly small. The smaller the signature, the less information Malwise has to compare binaries. In practice, this is becoming less of a problem as the size of malware binaries is increasing and malware becomes increasingly more complex.

### 5.2.6 Comparisons to Previous Work

Because the antivirus industry normally follows a closed research paradigm, most published work does not have a real working malware detection system for the research community to compare against. This is also the motivation and contribution of our work. We hope this paper can be a public benchmark available to the research community. The most useful comparison examining the effectiveness and efficiency of our work is in [9]. In their work, they constructed similarity matrices comparing variants such as the Netsky family of malware using the call graphs of each sample. We use the same evaluation conditions to test how many variants of Netsky family malware can be identified. We achieve 57 percent detection rate, compared to 40 percent detection rate in the previous work [9]. Malwise identifies a greater number of variants in the Netsky family than [9].

There are differences between our work and all the previous work. First, our work aims to discriminate between malicious and benign samples while the previous work aims to use the results only for clustering. Second, our work is completely automatic. Previous work used manual unpacking which may generate more accurate results. Therefore, we see the relationship between our work and previous work as being in agreement in being able to effectively identify malware. Additionally, our work performs in close to real time whereas in previous work an offline analysis was done with no quantitative analysis on efficiency.

## 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 different algorithms to unpack malware using application-level emulation. We also proposed performing malware classification using either the edit distance between structured control flow graphs, or the estimation of isomorphism between control flow graphs. We implemented and evaluated these approaches in a fully functional system, named Malwise. The automated unpacking was 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 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, it was demonstrated the efficiency of unpacking and malware classification warrants Malwise as suitable for potential applications including desktop and Internet gateway and antivirus systems.

## References


[1] "Symantec Internet Security Threat Report: Volume XII,"Symantec, 2008.

[2] "F-Secure Reports Amount of Malware Grew by 100 Percent during 2007,"F-Secure, http://www.f-secure.com/en_EMEA/about-us/pressroom/news/2007/fs_news_20071204_1_eng.html, 2007.

[3] K. Griffin, S. Schneider, X. Hu, and T. Chiueh, "Automatic Generation of String Signatures for Malware Detection," *Proc. 12th Int'l Symp. Recent Advances in Intrusion Detection (RAID '09)*, 2009.

[4] J.O. Kephart and W.C. Arnold, "Automatic Extraction of Computer Virus Signatures," *Proc. Int'l Conf. Fourth Virus Bull.*, pp. 178-184, 1994.

[5] J.Z. Kolter and M.A. Maloof, "Learning to Detect Malicious Executables in the Wild," *Proc. Int'l Conf. Knowledge Discovery and Data Mining*, pp. 470-478, 2004.

[6] M.E. Karim, A. Walenstein, A. Lakhotia, and L. Parida, "Malware Phylogeny Generation Using Permutations of Code," *J. Computer Virology*, vol. 1, pp. 13-23, 2005.

[7] M. Gheorghescu, "An Automated Virus Classification System," *Proc. Virus Bull. Conf.*, pp. 294-300, 2005.

[8] Y. Ye, D. Wang, T. Li, and D. Ye, "IMDS: Intelligent Malware Detection System," *Proc. 13th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, 2007.

[9] E. Carrera and G. Erdélyi, "Digital Genome Mapping-Advanced Binary Malware Analysis," *Proc. Virus Bull. Conf.*, pp. 187-197, 2004.

[10] T. Dullien and R. Rolles, "Graph-Based Comparison of Executable Objects (English Version)," *Proc. SSTIC*, 2005.

[11] I. Briones and A. Gomez, "Graphs, Entropy and Grid Computing: Automatic Comparison of Malware," *Proc. Virus Bull. Conf.*, pp. 1-12, 2008.

[12] S. Cesare and Y. Xiang, "Classification of Malware Using Structured Control Flow," *Proc. Eighth Australasian Symp. Parallel and Distributed Computing (AusPDC '10)*, 2010.

[13] G. Bonfante, M. Kaczmarek, and J.Y. Marion, "Morphological Detection of Malware," *Proc. IEEE Int'l Conf. Malicious and Unwanted Software*, pp. 1-8, 2008.

[14] R.T. Gerald and A.F. Lori, "Polymorphic Malware Detection and Identification via Context-Free Grammar Homomorphism," *Bell Labs Technical J.*, vol. 12, pp. 139-147, 2007.

[15] X. Hu, T. Chiueh, and K.G. Shin, "Large-Scale Malware Indexing Using Function-Call Graphs," *Proc. ACM Conf. Computer and Comm. Security*, pp. 611-620, 2009.

[16] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee, "Polyunpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware," *Proc. Computer Security Applications Conf.*, pp. 289-300, 2006.

[17] "Mal(ware)formation Statistics - Panda Research Blog,"Panda Research, http://research.pandasecurity.com/archive/Mal_2800_ware_2900_formation-statistics.aspx, 2009.

[18] A. Stepan, "Improving Proactive Detection of Packed Malware," *Proc. Virus Bull. Conf.*, 2006.

[19] M.G. Kang, P. Poosankam, and H. Yin, "Renovo: A Hidden Code Extractor for Packed Executables," *Proc. Workshop Recurring Malcode*, pp. 46-53, 2007.

[20] L. Boehne, "Pandora's Bochs: Automatic Unpacking of Malware," Diploma thesis, Univ. of Mannheim, 2008.

[21] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting System Emulators," *Proc. Information Security Conf.*, p. 1, 2007.

[22] L. Sun, T. Ebringer, and S. Boztas, "Hump-and-Dump: Efficient Generic Unpacking Using an Ordered Address Execution Histogram," *Proc. Int'l Computer Anti-Virus Researchers Organization (CARO) Workshop*, 2008.

[23] T. Graf, "Generic Unpacking: How to Handle Modified or Unknown PE Compression Engines," *Proc. Virus Bull. Conf.*, 2005.

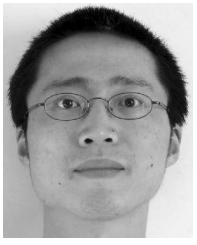
[24] D. Quist and Valsmith, "Covert Debugging Circumventing Software Armoring Techniques," *Proc. Black Hat Briefings*, 2007.

[25] L. Martignoni, M. Christodorescu, and S. Jha, "Omniunpack: Fast, Generic, and Safe Unpacking of Malware," *Proc. Ann. Computer Security Applications Conf. (ACSAC)*, pp. 431-441, 2007.

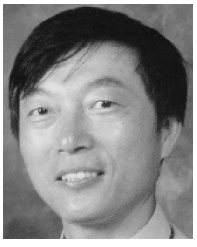
[26] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware Analysis via Hardware Virtualization Extensions," *Proc. 15th ACM Conf. Computer and Comm. Security*, pp. 51-62, 2008.

[27] R. Perdisci, A. Lanzi, and W. Lee, "McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables," *Proc. Ann. Computer Security Applications Conf.*, pp. 301-310, 2008.
[28] Y. Tang, B. Xiao, and X. Lu, "Signature Tree Generation for Polymorphic Worms," *IEEE Trans. Computers*, vol. 58, no. 4, pp. 565-579, Apr. 2011.
[29] M. Dalla Preda, R. Giacobazzi, S. Debray, K. Coogan, and G. Townsend, "Modelling Metamorphism by Abstract Interpretation," *Proc. Int'l Conf. Static Analysis*, R. Cousot and M. Martel, eds., pp. 218-235, 2011.
[30] Zynamics, VxClass, http://www.zynamics.com/vxclass.html, 2009.
[31] D. Gao, M.K. Reiter, and D. Song, "Binhunt: Automatically Finding Semantic Differences in Binary Programs," *Proc. Int'l Conf. Information and Comm. Security*, pp. 238-255, 2008.
[32] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic Worm Detection Using Structural Information of Executables," *Proc. Int'l Conf. Recent Advances in Intrusion Detection*, pp. 207-226, 2006.
[33] J. Kinable and O. Kostakis, "Malware Classification Based on Call Graph Clustering," *J. Computer Virology*, vol. 7, pp. 233-245, 2011.
[34] R. Lyda and J. Hamrock, "Using Entropy Analysis to Find Encrypted and Packed Malware," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 40-45, Mar./Apr. 2007.
[35] Y. Wu, T. Chiueh, and C. Zhao, "Efficient and Automatic Instrumentation for Packed Binaries," *Proc. Int'l Conf. and Workshops Advances in Information Security and Assurance*, pp. 307-316, 2009.
[36] I. Santos, X. Ugarte-Pedrero, B. Sanz, C. Laorden, and P.G. Bringas, "Collective Classification for Packed Executable Identification," *Proc. Eighth Ann. Collaboration, Electronic Messaging, Anti-Abuse and Spam Conf. (CEAS '11)*, pp. 23-30, 2011.
[37] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries," *Proc. USENIX Security Symp.*, p. 18, 2004.
[38] C. Cifuentes, "Reverse Compilation Techniques," PhD thesis, Queensland Univ. of Technology, 1994.
[39] E. Moretti, G. Chanteperdrix, and A. Osorio, "New Algorithms for Control-Flow Graph Structuring," *Proc. Conf. Software Maintenance and Reeng.*, 2001.
[40] T. Wei, J. Mao, W. Zou, and Y. Chen, "Structuring 2-Way Branches in Binary Executables," *Proc. Int'l Computer Software and Applications Conf.*, 2007.
[41] R. Baeza-Yates and G. Navarro, "Fast Approximate String Matching in a Dictionary," *Proc. South Am. Symp. String Processing and Information Retrieval (SPIR '98)*, pp. 14-22, 1998.
[42] A.V. Aho and M.J. Corasick, "Efficient String Matching: An Aid to Bibliographic Search," *Comm. ACM*, vol. 18, pp. 333-340, 1975.
[43] *Offensive Computing*, http://www.offensivecomputing.net, 2009.
[44] *Mwcollect Alliance*, http://alliance.mwcollect.org, 2009.

**Silvio Cesare** received the bachelor's degree in information technology and the master's degree in informatics from CQUniversity. He is currently working toward the PhD degree at Deakin University and has received the DUPRA scholarship. His research interests include analyzing software for theft, vulnerabilities, bugs, or malicious intent. He has previously been employed as a scanner architect in the network vulnerability assessment and management industry. He is a student member of the IEEE and the IEEE Computer Society.



**Yang Xiang** received the PhD degree in computer science from Deakin University, Australia. He is currently with the School of Information Technology, Deakin University. His research interests include network and system security, distributed systems, and networking. In particular, he is currently leading in a research group developing active defense systems against large-scale distributed network attacks. He is the chief investigator of several projects in network and system security, funded by the Australian Research Council. He has published more than 100 research papers in many international journals and conferences, such as *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Information Security and Forensics*, and *IEEE Journal on Selected Areas in Communications*. He has served as the program/general chair for many international conferences such as ICA3PP 2012/2011, IEEE/IFIP EUC 2011, IEEE TrustCom 2011, IEEE HPCC 2010/2009, IEEE ICPADS 2008, and NSS 2011/2010/2009/2008/2007. He has been the PC member for more than 50 international conferences in distributed systems, networking, and security. He serves as the associate editor of *IEEE Transactions on Parallel and Distributed Systems* and the editor of *Journal of Network and Computer Applications*. He is a senior member of the IEEE and the IEEE Computer Society.



**Wanlei Zhou** received the PhD degree from the Australian National University, Canberra, Australia, in 1991 and the DSc degree from Deakin University, Victoria, Australia, in 2002. He is currently the chair professor of information technology and the head of the School of Information Technology, Deakin University, Melbourne. His research interests include distributed and parallel systems, network security, mobile computing, bioinformatics, and e-learning. He has published more than 200 papers in refereed international journals and refereed international conference proceedings. Since 1997, he has been involved in more than 50 international conferences as the general chair, a steering chair, a PC chair, a session chair, a publication chair, and a PC member. He is a senior member of the IEEE and the IEEE Computer Society.