Published in the Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04), July 11-14, 2004, Boston, Massachusetts, USA.

# Testing Malware Detectors [*]

Mihai Christodorescu and Somesh Jha
Computer Sciences Department
University of Wisconsin
1210 W Dayton St
Madison, WI 53706, USA
{mihai,jha}@cs.wisc.edu

## ABSTRACT

In today's interconnected world, malware, such as worms and viruses, can cause havoc. A malware detector (commonly known as *virus scanner*) attempts to identify malware. In spite of the importance of malware detectors, there is a dearth of testing techniques for evaluating them. We present a technique based on program obfuscation for generating tests for malware detectors. Our technique is geared towards evaluating the resilience of malware detectors to various obfuscation transformations commonly used by hackers to disguise malware. We also demonstrate that a hacker can leverage a malware detector's weakness in handling obfuscation transformations and can extract the signature used by a detector for a specific malware. We evaluate three widely-used commercial virus scanners using our techniques and discover that the resilience of these scanners to various obfuscations is very poor.

## Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.5 [**Software Engineering**]: Testing and Debugging; D.4 [**Software**]: Operating Systems; D.4.6 [**Operating Systems**]: Security and Protection

## General Terms

Algorithms, experimentation, measurement, security.

## Keywords

Anti-virus, adaptive testing, malware, obfuscation.

[*]This work was supported in part by the Office of Naval Research under contracts N00014-01-1-0796 and N00014-01-1-0708. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notices affixed thereon. The views and conclusions contained herein are those of the authors, and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the above government agencies or the U.S. Government.

## 1. INTRODUCTION

Malicious code can infiltrate hosts using a variety of methods, such as attacks that exploit known software flaws, hidden functionality in regular programs, and social engineering. A *malware detector* identifies and contains malware before it can reach a system or network. A thorough evaluation of the quality of a malware detector has to take into account the wide variety of threats that malwares pose. A classification of malware with respect to its propagation method and goal is given in [26].

This paper addresses the problem of testing malware detectors. We focus on testing anti-virus software (such as commercial virus scanners), but our techniques are general and are applicable to other types of malware detectors. In order to understand the difficulties in testing malware detectors one has to understand the *obfuscation-deobfuscation game* that malware writers and developers of malware detectors play against each other. As detection techniques advance, malware writers use better hiding techniques to evade detection; in response, developers of malware detectors deploy better detection algorithms. For example, polymorphic and metamorphic viruses (and, more recently, shellcodes [9]) are specifically designed to bypass detection tools. A *polymorphic virus* "morphs" itself in order to evade detection. A common technique to "morph" viruses is to encrypt the malicious payload and decrypt it during execution. To obfuscate the decryption routine, several transformations, such as `nop`-insertion, code transposition (changing the order of instructions and placing jump instructions to maintain the original semantics), and register reassignment (permuting the register allocation), are applied. *Metamorphic viruses* attempt to evade heuristic detection techniques by using more complex obfuscations. When they replicate, these viruses change their code in a variety of ways, such as code transposition, substitution of equivalent instruction sequences, change of conditional jumps, and register reassignment [22, 33, 43]. Furthermore, they can "weave" the virus code into a host program, making detection by traditional heuristics almost impossible since the virus code is mixed with program code and the virus entry point is no longer at the beginning of the program (these are designated as entry point obscuring viruses [18]).

As hackers borrow and build on top of each other's techniques, malwares share significant code fragments. Frequently, a virus goes through a development cycle where each successive version is only slightly different from its predecessor. For example, the *Sobig.A* through *Sobig.F* worm variants (widespread during the summer of 2003) were developed incrementally, with each successive iteration adding or changing small features to the *Sobig* worm [19, 20, 21].

Given the obfuscation-deobfuscation game and the code reuse practiced by virus writers, two questions arise:

> **Question 1:** How resistant is a malware detector to obfuscations or variants of known malware?
>
> **Question 2:** Using limitations of a malware detector in handling obfuscations, can a hacker or a blackhat[1] determine its detection algorithm?

Question 1 is motivated by the obfuscation-deobfuscation game. Question 2 is motivated by the fact that if a blackhat knows the detection algorithm used by a malware detector, they can better target their evasion techniques. In other words, the "stealth" of the detection algorithm is important.

This papers addresses the above-mentioned questions and makes the following contributions:

- **Test-case generation using program obfuscation:** Motivated by protection of intellectual property, several researchers have studied program obfuscation to make reverse engineering of code harder [5]. We use transformations from the program obfuscation literature to generate test cases for malware detectors. We have developed an obfuscator (see Section 3) for Visual Basic programs. Given a known malware, we generate several variants of it by applying various obfuscation transformations. These variants are then treated as test cases for malware detectors. Our obfuscation-based testing technique is motivated by Question 1. Our testing methodology is described in Section 4.

- **Signature extraction:** Motivated by question 2, we have developed a signature-extraction algorithm which uses a malware detector, such as a virus scanner, as a black box. Using the output of a malware detector on several obfuscations of a malware, we extract signature used by the detection algorithm. This signature-extraction algorithm is described in Section 4.2.

- **Experimental evaluation:** Using our testing methodology we evaluated three commercial virus scanners: Symantec's Norton AntiVirus version 7.61.934 for Windows 2000, Network Associates Technology Inc.'s McAfee Virus Scan version 4.24.0 for Linux, and Sophos Plc's Sophos Antivirus version 3.76 (engine version 2.17) for Linux/Intel. The resilience of these virus scanners to various obfuscation transformations was quite varied. Results of our testing effort are summarized in Section 5. Using our signature-extraction algorithm, we were also able to extract signatures used by the virus scanners for various malware. The extracted signatures are presented in Section 5. From our experimental results we conclude that the state of the art for malware detectors is dismal!

## 2. RELATED WORK

The testing technique in this paper draws upon three major areas: testing of malware detectors, general software testing, and program obfuscation. In the following subsections we describe related work from these areas and highlight the contributions presented in this paper.

[1]"Hacker" and "blackhat" will be used synonymously throughout this paper.

### 2.1 Related work on testing malware detectors

As the anti-virus software industry ramped up in the early 1990s, the testing and reviewing procedures trailed the spread of viruses and the advances in detection algorithms. As late as 1996, Sarah Gordon opened her paper [13] on the state of affairs of anti-virus testing with the following observation:

> "The evaluation of anti-virus software is not adequately covered by any existing criteria based on formal methods. The process, therefore, has been carried out by various personnel using a variety of tools and methods."

Although there is no shortage of benign programs, the major difficulty in testing malware detectors is finding a suite of malicious programs. Motivated by this difficulty various researchers have categorized known malicious programs, $\mathcal{M}_{known}$, into malware that is in the wild ($ITW$), currently spreading and infecting new targets, and malware that is only found in online collections (the *Zoo*):

$$\mathcal{M}_{known} = \mathcal{M}_{ITW} \cup \mathcal{M}_{Zoo}$$

The set $\mathcal{M}_{ITW}$ represents the collection of malicious programs known to be in the wild, "spreading as a result of normal day-to-day operations on and between the computers of unsuspecting users" [35]. The set $\mathcal{M}_{Zoo}$ contains known malware that is not currently known to be in the wild, either because it is no longer infecting systems or because it is a lab sample that never spread. In 1995, Joe Wells started *The WildList*, a list of viruses reported to be in the wild over a certain period (currently it is updated monthly [35]). *The WildList* is an important test set for malware detectors, and established testing and certifications labs (e.g. ICSA [17], Checkmark [38, 39], and Virus Bulletin [36]) require malware detectors to identify all programs in $\mathcal{M}_{ITW}$ with a detection rate of 100% (based on a current version of *The WildList*) and at least 90% for malware in $\mathcal{M}_{Zoo}$. However, using $\mathcal{M}_{ITW}$ as a test set does not evaluate the resilience of malware detectors to obfuscation transformation, a common strategy used by blackhats.

Marx [23] presented a compendium of anti-virus program testing techniques and methods, with descriptions of the relevant metrics and the possible pitfalls that can invalidate the results. The question of assessing the unknown malware detection capabilities is still open, and Brunnstein [2] performed tests geared specifically at the heuristic detection algorithms that many malware detectors employ. Marx [24] extended this work by proposing the use of *retrospective testing* to measure the quality of the heuristics. Retrospective testing executes older versions of virus scanners against new malware not known at the time the virus scanners were released. To our knowledge, ours is the first paper that uses program obfuscation as a testing technique for malware detectors. Moreover, signature extraction has also not been addressed in the existing literature.

### 2.2 Related work on software testing

The software testing literature is rich in models and approaches. When choosing a testing methodology, one has to consider the nature of malware detectors: first, most malware detectors are commercial software, and their algorithms and technologies are proprietary. This limits us to the use of functional or black-box testing [29]. Second, the very nature of malware makes it impossible to define it clearly: a malware can be seen as code that bypasses the security policy of a system. This means that the input space cannot be simply reduced to a manageable size by using equivalence classes, such as "worms," "viruses," and "benign programs". We consider two methods for exploring the input space: random and adaptive testing.

*Random testing* generates tests by randomly selecting inputs from the domain of a given program. While intuitively considered a poor strategy for test-case generation, random testing is regarded by many as a viable approach. Compared to the general strategy of *partition testing* [30], which describes any approach to use information about the program to split the input domain into partitions and to generate test cases from each partition, random testing can perform as well [10] or better [14]. More recent work [3, 31, 40] proves that under certain conditions (if, for example, a partition contains only correct, non-fault inducing, inputs) random testing is a good approach, while partition testing is better under different scenarios (for example, when all the partitions have equal size). *Operational testing* [12], a variant of random testing, uses probable operational input (i.e. typical usage data) to generate test sets. In practice, *fuzz testing* [11, 27, 28], a form of random testing, has proved to be a powerful technique for generating test sets for programs. To our knowledge, the combination of program obfuscation and random testing for evaluating malware detectors has not been investigated before.

*Adaptive testing*, introduced by Cooper [8], describes a testing strategy that uses an objective function to guide the exploration of the input domain. By iteratively generating test sets based on evaluation of an objective function over the previous testing results, adaptive testing can identify the performance boundary of a given program. At each iteration step, the test set is generated by perturbing the previous set according to some algorithm that attempts to optimize (minimize or maximize) an objective function. Our signature-extraction algorithm bears some similarity to these adaptive-testing techniques. However, signature extraction has not been addressed in the literature before. The testing strategy described in this paper also combines features of operational testing with those of *debug testing* [12], where tests are designed to seek out failures. In this area, Hildebrandt and Zeller [15, 16] introduced a delta debugging algorithm that seeks to minimize failure-inducing input and runs in $O(n^2)$ time. We enhance their technique by employing knowledge about the input structure (all inputs to the malware detector are valid programs) and by modeling the malware detector, to achieve a running time of $O(k \log n)$ for the signature-extraction algorithm, where $k$ is the amount of information we learn about the signature used by the malware detector in one iteration.

## 2.3 Related work on program obfuscation

Obfuscation techniques have been the focus of much research due to their relevance to the protection of intellectual property present in software programs. The goal is to render a program hard to analyze and reverse engineer. Collberg, Thomborson, and Low [5] defined obfuscation as a program transformation that maintains the "observable behavior." They also introduced a taxonomy of obfuscation transformations based on three metrics: potency against a human analyst, resilience against an automatic tool, and execution overhead. Furthermore, they proposed control transformations using *opaque predicates* (predicates with values known at obfuscation time, but otherwise hard to analyze and statically compute) and computation and data transformations that break known algorithms and data structure abstractions while preserving semantics [6, 7]. Chow, Gu, Johnson, and Zakharov [4] presented an obfuscation approach based on inserting hard combinatorial problems with known solutions into the program using semantics-preserving transformations, which the authors claim make deobfuscation PSPACE-complete. Wang [37] introduced obfuscation as "one-way translation" in the context of a security architecture for survivability. Wroblewski's general method of obfuscation [41] uses code reordering and insertion in the context of semantics-preserving obfuscation transformations. To our knowledge, ours is the only paper which considers program obfuscation as a technique for testing malware detectors.

While deciding on the initial obfuscation techniques to focus on, we were also influenced by several existing blackhat tools. *Mistfall* (by *z0mbie*) is a library for binary obfuscation, specifically written to blend malicious code into a host program [42]. It can encrypt the virus code and data, obfuscate the virus control flow, and blend it into the host program. *Burneye* (by *TESO*) is a Linux binary encapsulation tool that encrypts a binary (possibly multiple times) and packages it into a new binary with an extraction tool [34].

## 3. PROGRAM OBFUSCATION

This section provides a definition of program obfuscation and our implementation of an obfuscator for Visual Basic.

An obfuscation transformation $\sigma$ is a program transformation that does not change the behavior of the original program, i.e., it adds code or transforms the existing code in such a way that it does not affect the overall result, or it adds new behaviors without affecting the existing ones. If we regard a program as a function that maps inputs to outputs, $p : \mathit{Inputs} \to \mathit{Outputs}$, then an obfuscation transformation $\sigma$ must conform to the following condition:

$$\forall I \in \mathit{Inputs} \,.\, p(I) \subseteq \big(\sigma(p)\big)(I)$$

Notice that for a specific input $I$ the obfuscated program $\sigma(p)$ allows a larger set of outputs than the original program $p$. This relaxed definition of obfuscation (as opposed to a strict semantics-preserving definition) is similar to the one introduced in [37] and has several key benefits in our context. First, when used for generating test cases, this relaxed definition models the evolution of actual malicious code better. This is based on the observation that new malware borrows from existing code, and many times successive versions of the same malware only add new behaviors. Second, in the case of viruses that infect executable binary files, we want to treat the program hosting the virus as a decoy, i.e., as an obfuscation that the virus uses to hide itself.

### 3.1 Our Implementation

Our obfuscator works on Visual Basic programs. First, we parse the program and construct various structures which are used by the obfuscation transformations. We chose three program structures as the bases for applying these transformations: the *abstract syntax tree (AST)* built from parsing the original program, the *control flow graph (CFG)* for each subroutine (procedure or function) in the program, and the list of subroutines. Other structures, such as call graphs, data dependence graphs, control dependence graphs, and system dependence graphs, are constructed on demand from these three data structures.

Transformations applied to the AST allow a detailed specification of the program layout on disk, where the order of instructions might be different from the execution order. Transformations applied to the CFG are defined as graph transformations, using node replacement grammars with a limited amount of context sensitivity provided by evaluating the truth value of static analysis predicates associated to each rewrite rule. Transformations applied to the list of subroutines relate to adding, removing, or replacing subroutines in the program. This type of transformation is in most cases applied in combination with CFG or AST transformations. For example, outlining (extracting code from an existing subroutine and creating a new subroutine with this code) involves both a set of CFG transformations and a subroutine addition.

Next, we describe four important obfuscations supported by our implementation. These obfuscations are inspired by existing real-world malware.

## 3.2 Sample obfuscations

We present four obfuscation transformations we implemented. Each program transformation is defined by its type and its associated parameters. The code example used in this section is derived from a fragment of the Homepage virus shown in Listing 1. The code in this listing implements the initial steps of the replication algorithm in the Homepage virus, by loading a copy of itself into a memory buffer and getting a handle to the system temporary directory.

```
On Error Resume Next
Set WS = CreateObject("WScript.Shell")
Set FSO= Createobject("scripting.filesystemobject")
Folder=FSO.GetSpecialFolder(2)

Set InF=FSO.OpenTextFile(WScript.ScriptFullname,1)
Do While InF.AtEndOfStream<>True
  ScriptBuffer=ScriptBuffer&InF.ReadLine&vbcrlf
Loop
```

Listing 1: Fragment of the Homepage virus, used in illustrating obfuscation transformations henceforth.

### 3.2.1 *Garbage Insertion*

Garbage insertion adds sequences which do not modify the behavior of the program. This insertion can be done anywhere in the code section of the program.

**Parameters**. The following parameters control this obfuscation transformation:

· **location** $\lambda$ represents the program point where the garbage insertion is to be performed;

· **size** $\sigma$ represents the size of the garbage code sequence to be generated for insertion.

Currently our implementation supports insertion of garbage code at any location in the program. The garbage code sequence is randomly generated from a combination of assignments (involving simple arithmetic) and branch statements with conditionals based on variables used only in the garbage code.

### 3.2.2 *Code Reordering*

The code reordering obfuscation changes the order of program instructions. The physical order is changed while maintaining the original execution order through the use of control-flow instructions (branches and jumps). Branches are inserted with conditionals defined and computed such that the branch is always taken. The conditional expression can be based on a complex computation.

The execution order of instructions can be changed only if the program behavior is not affected. Independent consecutive instructions (without any dependencies between them) can thus be interchanged.

**Parameters**. The following parameters control the code-reordering obfuscation:

· **program range** $(\lambda_{begin}, \lambda_{end})$ represents the set of instructions to be reordered;

· **type of reordering** $\tau$ selects between physical reordering, execution reordering, or both.

The reorder obfuscation transformation can process any statements in the program and physically reorder them either randomly or in an order strictly reversed from the original program order. Listing 2 shows an example of a reordering creating a strictly reversed order.

```
GoTo label_0001
label_0006:
  Do While InF.AtEndOfStream<>True
    ScriptBuffer=ScriptBuffer&InF.ReadLine&vbcrlf
  Loop
  GoTo label_0007
label_0005:
  Set InF=FSO.OpenTextFile(WScript.ScriptFullname,1)
  GoTo label_0006
label_0004:
  Folder=FSO.GetSpecialFolder(2)
  GoTo label_0005
label_0003:
  Set FSO= Createobject("scripting.filesystemobject")
  GoTo label_0004
label_0002:
  Set WS = CreateObject("WScript.Shell")
  GoTo label_0003
label_0001:
  On Error Resume Next
  GoTo label_0002
label_0007:
```

Listing 2: The result of a reorder obfuscation applied to the code fragment in Listing 1.

### 3.2.3 *Variable Renaming*

The renaming obfuscation transformation replaces a given identifier with another. The replacement can occur throughout a given live range of the variable, or throughout the whole program.

**Parameters**. The following parameters control the renaming obfuscation:

· **name** $\nu_{old}$ is the name of the variable to be replaced;

· **new name** $\nu_{new}$ is the new identifier meant to replace $\nu_{old}$;

· **program location** $\lambda$ identifies a program location part of the live range of $\nu_{old}$ over which the replacement is to take place.

Our implementation can rename any variable in the program, either for a limited range of statements or throughout its live range. The new names are randomly picked from the system dictionary, as illustrated in the example shown in Listing 3, where all the variables are renamed throughout the program body.

```
On Error Resume Next
Set inquiries = CreateObject("WScript.Shell")
Set will= Createobject("scripting.filesystemobject")
grimier=will.GetSpecialFolder(2)

Set rumour=will.OpenTextFile(WScript.ScriptFullname,1)
Do While rumour.AtEndOfStream<>True
  optimizers=optimizers&rumour.ReadLine&vbcrlf
Loop
```

Listing 3: The result of a variable renaming obfuscation applied to the code fragment in Listing 1.

### 3.2.4 Code and Data Encapsulation

The encapsulation obfuscation is similar to a self-extracting file archive. The chosen program fragment is encoded using a specified algorithm, and a decoding procedure is inserted in the program. At execution time, when the encapsulated program fragment is needed, the decoding procedure is executed and the original program fragment is used, i.e., executed if it is a code portion, or accessed if it is a data portion. This obfuscation is designed to be used with run-time immutable program portions (e.g. code of non-self-modifying programs, and read-only data).

**Parameters**. The following parameters control the encapsulation obfuscation transformation:

· **program range** $(\lambda_{begin}, \lambda_{end})$ represents the program portion to be encapsulated;

· **type of encapsulation** $\tau$ specifies the technique used to transform the program range;

· other parameters specific to the type of encapsulation.

The type of encapsulation $\tau$ selects the algorithm used to transform the sequence of bits representing the program fragment. The type of encapsulation can range from simple uuencoding, to any compression technique (e.g. ZIP), and to any encryption technique (e.g. RSA).

The encapsulation obfuscator currently offers two types of encodings: the *identity encoding* and a *hex encoding*. The identity encoding converts the code to a string and wraps it with a call to an interpreter (in Visual Basic, this is achieved using the `Execute()` function). A hex encoding replaces each byte in the original program portion with its ASCII representation (in hexadecimal) and passes the resulting string to the interpreter. The implementation is modular, allowing for additional encodings to be easily plugged in. A sample of the hex encoding is shown in Listing 4.

## 4. TESTING METHODOLOGY

A malware detector $D$ works by analyzing a data object (a file, an email message, or a network packet) and determining whether the data contains an executable and whether the executable is malicious. The first test performed by the malware detector (to determine the presence of executable content) is usually based on the host operating system's method for discovering the type of the data. The type can be determined based on MIME type headers, file extensions, or a "magic number" that is unique to a file format. Given that techniques exist to determine whether a data object contains an executable, we restrict our definition of a malware detector to admit only executable programs as inputs.

```
Execute( hex_decode( "4F6E204572726F7220526573" &
                     "756D65204E6578740A536574" &
                     ...
                     "66657226496E462E52656164" &
                     "4C696E6526766263726C660A" &
                     "4C6F6F700A" ) )

Function hex_decode( S )
  HexDecoder=""
  For I = 1 To Len( S ) Step 2
    dOne = CInt( "&H" & Mid( S, I, 1 ) )
    dTwo = CInt( "&H" & Mid( S, I + 1, 1 ) )
    hex_decode = hex_decode&Chr( dOne * 16 + dTwo )
  Next
End Function
```

Listing 4: The result of a hex encoding obfuscation applied to the code fragment in Listing 1.

We define a malware detector $D$ as a function whose domain and range are the set of executable programs $\mathcal{P}$ and the set $\{malicious, benign\}$, respectively. In other words, a malware detector $D$ is a function $D : \mathcal{P} \to \{malicious, benign\}$ defined as:

$$D(p) = \begin{cases} \text{malicious} & \text{if } p \text{ contains malicious code} \\ \text{benign} & \text{otherwise} \end{cases}$$

Testing a detector $D$ means iterating over all input programs $p \in \mathcal{P}$ and checking the correctness of the answer. In this context, *false positives* are benign programs that the malware detection tool marks as infected. *False negatives* are malwares that the detection tool fails to recognize. Conversely, the *hit rate* measures the ratio of malicious programs detected out of the malwares used for testing. In testing a malware detector, the goal is to verify whether the tool detects all malware. Given the potential threat posed by malware, it is critical to reduce the number of false negatives to be as close to zero as possible, since each false negative represents an undetected malicious program that is loose in the system. On the other hand, the number of false positives is important in determining the usability of the malware detector: if it incorrectly flags too many benign programs as infected, the user may lose faith in the malware detector and stop using it altogether.

Since the set $\mathcal{P}$ of all possible programs is infinite, simply enumerating all inputs for the malware detector is not possible. Every test set is thus finite, and the false positive, false negative, and hit rates are defined with respect to a given test set. A test set $\mathcal{P}_T$ is classified into two disjoint sets, one of benign programs $\mathcal{B}$ and one of malicious programs $\mathcal{M}$. The false positive rate $\text{FP}_{\mathcal{P}_T}$, false negative rate $\text{FN}_{\mathcal{P}_T}$, and hit rate $\text{HR}_{\mathcal{P}_T}$ (all relative to the test set $\mathcal{P}_T$) are defined as follows:

$$\text{FP}_{\mathcal{P}_T} \overset{\text{def}}{=} \frac{|\,\{p \in \mathcal{B} \;:\; D(p) = malicious\}\,|}{|\mathcal{B}|}$$

$$\text{FN}_{\mathcal{P}_T} \overset{\text{def}}{=} \frac{|\,\{p \in \mathcal{M} \;:\; D(p) = benign\}\,|}{|\mathcal{M}|}$$

$$\text{HR}_{\mathcal{P}_T} \overset{\text{def}}{=} \frac{|\,\{p \in \mathcal{M} \;:\; D(p) = malicious\}\,|}{|\mathcal{M}|}$$

The goal of the testing process is to measure a malware detector's false positive, false negative, and hit rates for a test set $\mathcal{P}_T$ and provide a measure of the detection efficacy.

As with any other testing procedure, the final assessment of the quality of a malware detector depends on the quality of the test set and the metrics that reflect the behavior of the program against the test inputs. Various testing techniques for evaluating malware detectors differ in their method for generating the test set $\mathcal{P}_T$. We describe a technique for generating tests for malware detectors which is based on program obfuscation.

## 4.1 Generating tests using program obfuscation

For our testing effort, we focus on malware. Therefore, we only report the false negative and hit rate of various malware detectors. We use obfuscation transformations applied to known malware to obtain a large number of new test programs that are semantically equivalent to the original malware – thus, the new test samples are just as malicious as the original malware and should be flagged by the detector. In other words, our test set $\mathcal{P}_T$ is created by applying semantics-preserving obfuscation transformations to a set of known malware. Our testing technique is geared towards evaluating the resilience of malware detectors to obfuscation transformations. This testing technique answers question 1 raised in the introduction.

The testing proceeded as follows: we collected 8 viruses and checked them against 3 commercial virus scanners. We made sure all the viruses were active and detected in their original form by these commercial virus scanners. To obtain our test set, we applied obfuscations from our implementation to the 8 viruses (see Section 3 for description of the obfuscation transformations). The parameters for each obfuscation were randomly varied to obtain multiple variants of each virus (unless the number of variants was naturally limited, e.g. renaming is constrained by the number of variables in the program). This approach follows the random testing technique, as we sample the space of obfuscated variants, instead of extensively enumerating all possible variants for each virus and each obfuscation – time consuming, and impossible in some cases. The number of variants generated for each virus were between 31 and 5,592. As our interest is in evaluating multiple malware detectors, we use the same test set for each detector. Results of our testing appears in Section 5.1.

The architecture of our testing toolkit is shown in Figure 1. The *obfuscation engine* applies obfuscation transformations according to specified parameters. The *result analyzer* records the output of the malware detector being tested. The *obfuscation parameter generator* determines the next set of parameters.

## 4.2 Signature Extraction

We now address the second question from the introduction, i.e., using limitations of a malware detector in handling obfuscations, can a blackhat determine the detection algorithm? Assume that the malware detector uses a signature to detect malware (this is true of most commercial virus scanners), i.e., a malware is identified by a sequence of statements called the *signature*. We introduce an iterative algorithm for discovering the signature used by a virus scanner to detect a specific malicious code. Our algorithm uses the malware detector as a black box and iteratively inputs obfuscated variants of a malware to the detector. We exploit the fact that most commercial virus scanners are

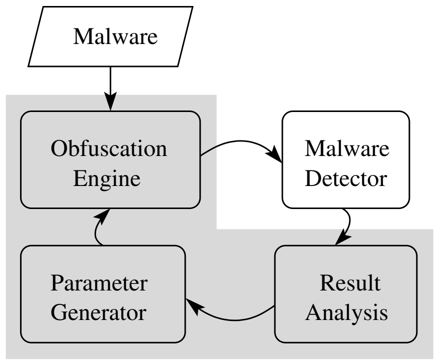

Figure 1: The architecture of the malicious code detector test toolkit.

not resistant to the encapsulation obfuscation transformation (see Section 3.2.4 for a description of the encapsulation transformation). We evaluated the three commercial virus scanners against our signature-extraction algorithm. Details of our evaluation can be found in Section 5.2. However, the following conclusions can be drawn from our evaluation:

- Weakness in handling obfuscations (in this case encapsulation) can be successfully used by a blackhat to extract signatures used for detection.
- Most commercial virus scanners are highly susceptible to our signature-extraction algorithm. Using our algorithm we were able to easily extract signatures.

Next, we provide details of our signature-extraction algorithm. We denote by $\textsc{Encode}(P, \{\lambda_1, \dots, \lambda_k\})$ the encapsulation obfuscation using the hex encoding applied to $k$ locations $\lambda_1, \dots, \lambda_k$ of a program $P$. We assume that the fragment of a malware that is encoded is "opaque" to the virus scanner, i.e., if we apply $\textsc{Encode}(P, \{\lambda_1, \dots, \lambda_k\})$, then the virus scanner does not know the contents of locations $\lambda_1, \dots, \lambda_k$. While the signature extraction algorithm presented uses the hex encoding obfuscation transformation, it does not depend on the specific type of obfuscation. The only requirement is that there exists one "opaque" obfuscation transformation for each malware detector–malware combination. During our experiments (Section 5.2), we we were able to create "opaque" encapsulation transformations without any difficulty. Suppose we input to the detector $D$ the malware $M$ with the encapsulation transformation $\textsc{Encode}$ applied to it. In this case, the detector $D$ will answer *malicious* if and only if the signature $\Sigma_{M,D}$ does not overlap with the encapsulated fragment $\lambda_1, \cdots, \lambda_k$. This fact is used in our signature-extraction algorithm and motivates our definition of a signature.

Definition 1. Malware signature
*Given a malware sample $M = \langle m_1, \dots, m_n \rangle$ of $n$ instructions and a malware detector $D$, a signature $\Sigma_{M,D}$ represents the minimal set of instructions such that:*

$$D(\textsc{Encode}(M, A)) = \begin{cases} \textit{benign} & \textit{if } A \subseteq \Sigma_{M,D} \ \wedge \ A \neq \emptyset \\ \textit{malicious} & \textit{otherwise} \end{cases}$$

*where $A = \{\lambda_1, \cdots, \lambda_k\}$ is the set of program locations encoded.*

The signature-extraction problem can be stated as follows:

Definition 2. The signature-extraction problem
*Given a malware sample $M$ and black-box access to a detector $D$, find the malware signature $\Sigma_{M,D}$ used by $D$ to identify $M$.*

Algorithm 1 describes our signature-extraction procedure. The core of the algorithm consists of a repeated binary search over the malware, at each step identifying the outermost (left and right) signature components. At the next iteration step, the binary search continues in the program range defined exclusively by the outermost signature components. After the first iteration of the **while** loop $L_1$ and $R_1$ are the lowest and the highest index of the signature $\Sigma_{M,D}$ in the malware $M = \langle m_1, \cdots, m_k\rangle$. The next iteration of the **while** loop restricts the search to the fragment $\langle m_{L_i+1}, \cdots, m_{R_i-1}\rangle$ of the malware $M$.

The procedure FindLeftmost finds the leftmost index of the signature in the malware, as illustrated in Algorithm 2. The notation $\mathcal{O}(V)$ represents a query to the malware detector on $V$ that returns the name of the detected malware (if any), and Encode$(M, L, R)$ is the encapsulation obfuscation transformation using hex encoding applied to the range $[L \ldots R]$ of the program $M$. The procedure FindRightmost (which finds the rightmost index of the signature) is similar, with a search biased towards the right half of the search range.

```
Input: A malware sample $M = \langle m_1, \ldots, m_n\rangle$ with $n$
       instructions, a malware detector $D$, and a malware name $\sigma$.
Output: The signature given as a set of malware instructions
        $\{m_{k_0}, \ldots, m_{k_i}, \ldots\}_{1 \le k_i \le n}$.

SignatureExtraction($M$, $D$, $\sigma$)
begin
    $L_0 \leftarrow 0$
    $R_0 \leftarrow N + 1$
    $i \leftarrow 1$
    while $L_{i-1} \neq 0 \wedge R_{i-1} \neq 0$ do
        $L_i \leftarrow$ FindLeftmost$(M, L_{i-1}+1, R_{i-1}-1, \sigma)$
        $R_i \leftarrow$ FindRightmost$(M, L_{i-1}+1, R_{i-1}-1, \sigma)$
        $i \leftarrow i + 1$
    end
    return $\{ m_k : \exists\, j < i \,.\, k = L_j \vee k = R_j \}$
end
```

**Algorithm 1:** Algorithm to extract the signature used by $D$ to detect $M$ as $\sigma$.

If we denote by $k$ the size of the malware signature (i.e. $|\Sigma_{M,D}| = k$) and assume that each query to the malware detector takes unit time, then the running time of our algorithm is $O(k \log n)$. During our experiments, we discovered that in most cases $k \ll n$, which means that the search quickly converges to find a signature. The average-case complexity of our algorithm is significantly better than the $O(k \log n)$ worst-case complexity. However, due to space limitations, we will not provide the derivation of the average-case complexity.

A malware detector usually uses multiple techniques to identify malicious code. These detection techniques form a hierarchy of signatures, and the detector first searches for the most specific signature and then works its way up the hierarchy. We have extended our algorithm to extract hierarchical signatures. For conciseness, we discuss only results based on the Algorithms 1 and 2.

```
Input: A malware sample $M = \langle m_1, \ldots, m_n\rangle$ with $n$
       instructions, a program range $[L..R]$, and a malware name $\sigma$.
Output: The leftmost index of a signature component
        within the given range.

FindLeftMost($M$, $L$, $R$, $\sigma$)
begin
    $sig \leftarrow 0$
    while $L \neq R$ do
        $C \leftarrow \frac{L+R}{2}$
        $V \leftarrow$ Encode$(M, L, C)$
        if $\mathcal{O}(V) = \sigma$ then $L \leftarrow C$
        else $R \leftarrow C$
    end
    $V \leftarrow$ Encode$(M, L, R)$
    if $\mathcal{O}(V) \neq \sigma$ then $sig \leftarrow L$
    return $sig$
end
```

**Algorithm 2:** Algorithm to find the leftmost signature component.

## 5. EXPERIMENTAL RESULTS

We applied our testing methodology to several Visual Basic malware. Due to space limitations we only report results on 8 malwares (Anna Kournikova, Homepage, Melissa, Tune, Chantal, GaScript, Lucky2, and Yovp) that exhibit various infection methods. For detailed descriptions of these malware, we refer the reader to the Symantec Antivirus Research Center's online virus database [32] and the McAfee AVERT Virus Information Library [25]. We used 3 commercial virus scanners in our tests: Symantec's Norton AntiVirus version 7.61.934 for Windows 2000, Network Associates Technology Inc.'s McAfee Virus Scan version 4.24.0 for Linux, and Sophos Plc's Sophos Antivirus version 3.76 (engine version 2.17) for Linux. All the virus scanners had their signature database updated before the tests were performed.

Our testing effort was successful. We gained information about the features of individual detection algorithms (for example, we learned that McAfee can detect malware very well in the presence of renaming obfuscation transformations). Our results suggest directions where improvements are needed. For example, code reordering and encapsulation obfuscations generate much higher false negative rates than renaming and garbage insertion obfuscations. Furthermore, we were able to discover many of the signatures used by the malware detectors and correlate their properties (such as signature size and specificity) with our random testing results.

**Obfuscation characteristics.** We investigate the tests generated by randomly applying various obfuscation transformations. Table 1 lists the original sizes of the malware along with minimum, average and maximum size of their obfuscated variants. In most cases, the size does increase (since we implemented obfuscation transformations that add additional code), but only by a limited amount – this means that in a real world scenario, the variants could spread just as easily as the original viruses, without imposing extra load on the network. There are several cases where the size of the variant is smaller than the size of the original malware. However, this only happens in the case of the renaming obfuscation, i.e., if the new names are shorter than the original names of the variables.

| *Malware name* | *Original size* | *Variant size* | | |
|---|---|---|---|---|
| | | *Min* | *Avg* | *Max* |
| Anna K. | 2,824 B | 110.02% | 126.47% | 144.04% |
| Homepage | 1,983 B | 118.83% | 156.17% | 193.71% |
| Melissa | 4,245 B | 95.64% | 121.81% | 151.39% |
| Tune | 7,003 B | 113.13% | 130.77% | 160.47% |
| Chantal | 417 B | 119.18% | 222.61% | 291.37% |
| GaScript | 3,568 B | 75.28% | 97.25% | 118.61% |
| Lucky2 | 686 B | 100.00% | 182.94% | 251.31% |
| Yovp | 1,223 B | 101.80% | 159.87% | 210.79% |

Table 1: The effect of the obfuscation transformations on malware sizes.

## 5.1 Testing resilience to obfuscations

We show the results of our random testing using obfuscated variants in Figures 2 and 3. Since we only tested on malwares, we do not report the false positive rate. Note that hit rate is simply one minus the false negative rate. The reader is warned against using these results to directly compare the three virus scanners, as they do not represent a complete assessment of their respective strengths and weaknesses. Our intent is to show that our testing techniques can expose enough information to discriminate between these virus scanners. From the results, it is immediately evident that our automatically-generated test sets provide enough data to make judgments of the relative strengths and weaknesses of each scanner with respect to handling obfuscations. For example, from Figure 2 we can deduce that the McAfee Virus Scanner handles variable renaming very well, while Norton AntiVirus can thwart code-reordering obfuscations.

It is somewhat surprising that for a given anti-virus tool the false negative rates (and, conversely, the hit rates) vary wildly across the malware set (see Figure 3). This leads us to believe that malware are identified using different techniques (some precise, some heuristic) that cope with obfuscations with varying degrees of success. Determining the actual detection techniques requires more data and finer-grained obfuscation techniques and will be investigated in the future. Another interesting result is that detection results are equally poor for code reordering and encapsulation obfuscation transformations. While encapsulation requires advanced detection mechanisms, such as emulation, sandboxing, and partial evaluation necessary to make the encoded fragments "visible", code reordering can be detected through simple traversals of the control-flow graph. Therefore, we can conclude that current anti-virus tools incorrectly assume the order of instructions in the malware to be the same as the execution order.

## 5.2 Signature extraction results

As shown in Table 3, our signature-extraction algorithm was successful in many cases. Due to space limitations

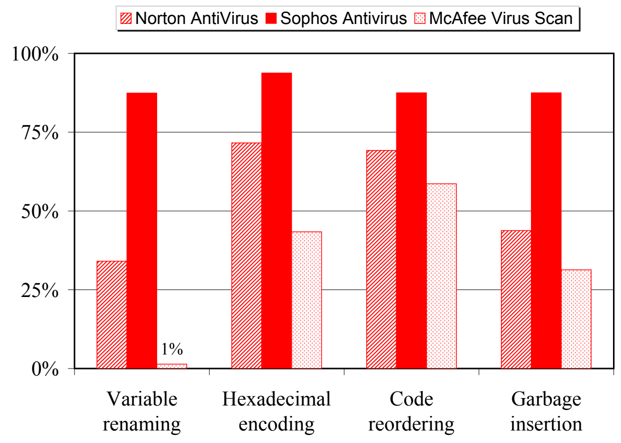

Figure 2: False negative rate for individual obfuscation transformations, averaged over the malware test set.

we only show the results for four malware samples. Results of our signature-extraction algorithm on other malware were similar. This proves that an adaptive-testing method, guided by the answers provided through black-box access to the malware detector, is successful in identifying signatures[2].

In some cases, the discovered signatures were single statements, while for other detectors the malware signatures consisted of multiple statements. Larger signatures reduce the number of false positives, since there is a smaller chance a benign program will contain the signature, but they are also less resilient to obfuscation attacks (see Table 2). We describe below several facts we learned from our testing results. These results demonstrate the richness of information that can be extracted about malware detectors using adaptive testing techniques.

| | Norton AntiVirus | | Sophos Antivirus | | McAfee Virus Scan | |
|---|---|---|---|---|---|---|
| | *sig %* | *fn %* | *sig %* | *fn %* | *sig %* | *fn %* |
| Anna K. | 3% | 12% | 41% | 12% | 100% | 75% |
| Melissa | 100% | 87% | 100% | 100% | 23% | 5% |
| Lucky2 | 6% | 85% | 100% | 100% | 22% | 53% |
| Yovp | 7% | 56% | 100% | 100% | 20% | 38% |

Table 2: The correlation between virus signature size (*sig %* is the size of the signature relative to the virus text) and the false negative rate (*fn %*).

**Signatures vary widely.** A quick look through Table 3 shows that each anti-virus vendor uses a different signature for a given virus. In some case there are overlaps between signatures from different vendors, e.g., Norton AntiVirus and Sophos Antivirus both refer to the line `Execute e7iqom5JE4z(...)` for the Anna Kournikova virus.

**Signatures target particular types of malicious activ-**

[2]During the editing of this paper, after filling out Table 3 and saving the document to disk, the anti-virus tool installed on one of the authors' machine identified this section of the LaTeX document as infected and quarantined it!

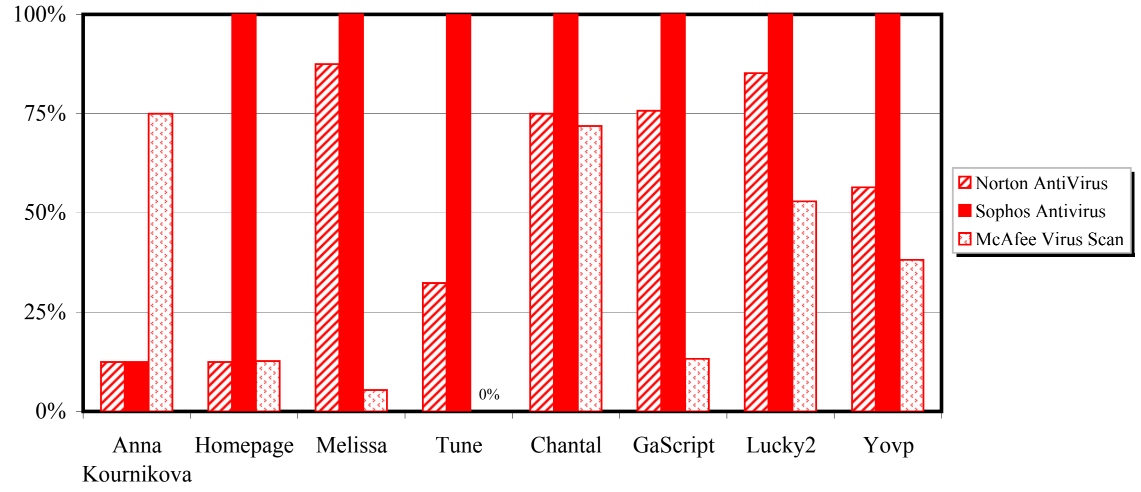

Figure 3: False negative rate for individual viruses, averaged over the set of obfuscation transformations.

**ities.** Several signatures discovered from our experiments clearly contain code describing a particular trait of the malware. For example, the McAfee Virus Scan signature for Lucky2 contains the code that replicates the virus by creating a copy of itself on the filesystem. Similarly, the Norton AntiVirus signature for Yovp captures the code that replicates the virus through floppy disks.

**Some signatures cover the whole virus body.** When the whole malware body is used as a signature, the signature-extraction algorithm could not identify any individual statements as being part of the signature. Such signatures are most precise in detecting a specific instance of the malware (reducing the false positive rate), but fail to match when the malware code is slightly obfuscated, thus increasing the false negative rate. This is supported in our experimental data by the observed correlation between whole virus body signatures and high false negative rates. For example, our signature extractor indicates that Sophos Antivirus uses the whole virus body as a signature for Melissa, Lucky2, and Yovp. Correspondingly, the false negative rates for these detector-virus pairs are high (100% in Figure 3).

**Some signatures are case-sensitive.** During the development of our toolkit, we discovered that in certain signatures the case of keywords in Visual Basic appears to be significant. The Microsoft Visual Basic interpreter is case-insensitive with respect to keywords. For certain virus scanners, changing one letter from uppercase to lowercase resulted in the virus not being detected. We intend to further pursue this issue in order to explore the limitations of signature-based virus detectors.

**Variable renaming handled well.** In our tests, the renaming obfuscation transformation did not pose great problems for malware detectors. Specifically, the McAfee Virus Scanner detected almost all variants mutated through variable renaming. When correlating whole virus body signatures with the random testing results, the resilience to variable renaming is the only feature that ameliorated a high false negative rate.

## 6. CONCLUSIONS AND FUTURE WORK

We present a novel obfuscation-based technique for testing malware detectors. Our testing shows that commercial virus scanners are not resilient to common obfuscations transformations. We also present a signature-extraction algorithm that exploits weaknesses of malware detectors in handling obfuscations. However, this paper opens several directions for further research. As shown in Section 5.2, a lot can be learned about a malware detector by using obfuscated variants of a known malware and by guiding the obfuscation to obtain the desired information. A logical next step will be to use more refined techniques once the malware signature is discovered. We will explore techniques from the learning regular languages [1] literature to design better signature-extraction algorithms. Obfuscations, such as variable renaming, encapsulation of string literals, and reordering, can provide more detailed information about a malware detector. For example, one might inquire whether the signature found is resilient to renaming transformations, or which parts of the signature can withstand encapsulation. The infrastructure for implementing these refinements is in place, so the research focus will be on finding algorithms to guide the refinement of the signature-extraction algorithm.

Another direction for future work is to explore the application of our testing methodology to binary programs, specifically the Intel *x86* platform. There is no inherent problem in using the algorithms presented in this paper on *x86* binaries – all the obfuscation transformations described are applicable to binary programs. We are in the process of developing a binary-rewriting toolkit that will allow us to implement these transformations and to test malware detectors using a larger test suite that includes both *x86* binary and Visual Basic malware.

## 7. REFERENCES

[1] D. Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 75:87–106, 1987.

| *Malware name* | *Malware detector* | *Extracted virus signature* |
|---|---|---|
| Anna Kournikova | Norton AntiVirus | `Execute e7iqom5JE4z("X)udQ0VpgjnH...70d2")` |
| | Sophos Antivirus | 1 `Execute e7iqom5JE4z("X)udQ0VpgjnH...70d2")`<br>5 `StTP1MoJ3ZU = Mid( hFeiuKrcoj3, I, 1 )`<br>6 `WHz23rBqlo7 = Mid( hFeiuKrcoj3, I + 1, 1 )`<br>8 `StTP1MoJ3ZU = Chr( 10 )`<br>10 `StTP1MoJ3ZU = Chr( 13 )`<br>12 `StTP1MoJ3ZU = Chr( 32 )`<br>14 `StTP1MoJ3ZU = Chr( Asc( StTP1MoJ3ZU ) - 2 )`<br>18 `WHz23rBqlo7 = Chr( 10 )`<br>20 `WHz23rBqlo7 = Chr( 13 )`<br>22 `WHz23rBqlo7 = Chr( 32 )`<br>24 `WHz23rBqlo7 = Chr( Asc( WHz23rBqlo7 ) - 2 )`<br>27 `e7iqom5JE4z = e7iqom5JE4 & WHz23rBqlo7 & StTP1MoJ3ZU` |
| | McAfee Virus Scan | *The whole body of the malware.* |
| Melissa | Norton AntiVirus | *The whole body of the malware.* |
| | Sophos Antivirus | *The whole body of the malware.* |
| | McAfee Virus Scan | *23 statements from the malware body.* |
| Lucky2 | Norton AntiVirus | `FSO.CopyFile Melhacker, target.Name, 1` |
| | Sophos Antivirus | *The whole body of the malware.* |
| | McAfee Virus Scan | 1 `Dim Melhacker, WshShell, FSO, VX, VirusLink`<br>6 `Melhacker = Wscript.ScriptFullName`<br>7 `VX = Left( Melhacker, InStrRev( Melhacker, "\" ) )`<br>9 `FSO.CopyFile Melhacker, target.Name, 1` |
| Yovp | Norton AntiVirus | 12 `dosfile.writeline( "command /f /c copy C:\viruz.vbs A:\" )`<br>13 `dosfile.writeline( "del C:\viruz.vbs" )` |
| | Sophos Antivirus | *The whole body of the malware.* |
| | McAfee Virus Scan | 1 `On Error Resume Next`<br>2 `Dim fso, wsh, dosfile, openvir, copyov`<br>3 `Set fso = createobject( "scripting.filesystemobject" )`<br>6 `Set dosfile = fso.createtextfile( "c:\dosfile.bat", true )`<br>7 `dosfile.writeline( "echo off" )`<br>8 `dosfile.writeline( "cd %windir%" )` |

Table 3: Signatures discovered by our signature-extraction algorithm. Where relevant, line numbers for each statement are provided.

[2] K. Brunnstein. "Heureka-2" AntiVirus Tests. Virus Test Center, University of Hamburg, Computer Science Department, Mar. 2002. Published online at http://agn-www.informatik.uni-hamburg.de/vtc/en0203.htm. Last accessed: 16 Jan. 2004.

[3] T. Chen and Y. Yu. On the relationship between partition and random testing. *IEEE Transactions on Software Engineering*, 20(12):977–980, Dec. 1994.

[4] S. Chow, Y. Gu, H. Johnson, and V. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In G. Davida and Y. Frankel, editors, *Proceedings of the 4th International Information Security Conference (ISC'01)*, volume 2200 of *Lecture Notes in Computer Science*, pages 144–155, Malaga, Spain, Oct. 2001. Springer-Verlag.

[5] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, July 1997.

[6] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proceedings of the International Conference on Computer Languages 1998 (ICCL'98)*, pages 28–39, Chicago, IL, USA, May 1998. IEEE Computer Society.

[7] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98)*, San Diego, CA, USA, Jan. 1998. ACM Press.

[8] D. W. Cooper. Adaptive testing. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE'76)*, pages 102–105, San Francisco, CA, USA, Oct. 1976. IEEE Computer Society Press.

[9] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. S. von Underduk. Polymorphic shellcode engine using spectrum analysis. *Phrack*, 11(61), Aug. 2003. Published online at http://www.phrack.org. Last accessed: 16 Jan. 2004.

[10] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(7):438–444, July 1984.

[11] J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 59–68, Seattle, WA, USA, Aug. 2000.

[12] P. G. Frankl, R. G. Hamlet, B. Littlewood, and L. Strigini. Choosing a testing method to deliver

reliability. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 68–78, Boston, MA, USA, May 1997.

[13] S. Gordon and R. Ford. Real world anti-virus product reviews and evaluations – the current state of affairs. In *Proceedings of the 19th National Information Systems Security Conference (NISSC'96)*, pages 526–538, Baltimore, MD, USA, Oct. 1996. National Institute of Standards and Technology (NIST).

[14] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1441, Dec. 1990.

[15] R. Hildebrandt and A. Zeller. Simplifying failure-inducing input. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 2000 (ISSTA'00)*, pages 135–145, Portland, OR, USA, 2000. ACM Press.

[16] R. Hildebrandt and A. Zeller. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, Feb. 2002.

[17] ICSA Labs. Anti-virus product certification. Published online at http://www.icsalabs.com/html/communities/antivirus/certification.shtml. Last accessed: 16 Jan. 2004.

[18] E. Kaspersky. *Virus List Encyclopedia*, chapter Ways of Infection: Viruses without an Entry Point. Kaspersky Labs, 2002.

[19] LURHQ Threat Intelligence Group. Sobig.a and the spam you received today. Technical report, LURHQ, 2003. Published online at http://www.lurhq.com/sobig.html. Last accessed on 16 Jan. 2004.

[20] LURHQ Threat Intelligence Group. Sobig.e - Evolution of the worm. Technical report, LURHQ, 2003. Published online at http://www.lurhq.com/sobig-e.html. Last accessed on 16 Jan. 2004.

[21] LURHQ Threat Intelligence Group. Sobig.f examined. Technical report, LURHQ, 2003. Published online at http://www.lurhq.com/sobig-f.html. Last accessed on 16 Jan. 2004.

[22] A. Marinescu. Russian doll. *Virus Bulletin*, pages 7–9, Aug. 2003.

[23] A. Marx. A guideline to anti-malware-software testing. In *Proceedings of the 9th Annual European Institute for Computer Antivirus Research Conference (EICAR'00)*, 2000.

[24] A. Marx. Retrospective testing – how good heuristics really work. In *Proceedings of the 2002 Virus Bulletin Conference (VB2002)*, New Orleans, LA, USA, Sept. 2002. Virus Bulletin.

[25] McAfee AVERT. Virus information library. Published online at http://us.mcafee.com/virusInfo/default.asp. Last accessed: 16 Jan. 2004.

[26] G. McGraw and G. Morrisett. Attacking malicious code: report to the Infosec research council. *IEEE Software*, 17(5):33 – 41, Sept./Oct. 2000.

[27] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):12–44, Dec. 1990.

[28] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services. Technical Report 1268, University of Wisconsin, Madison, Computer Sciences Department, Madison, WI, USA, Apr. 1995.

[29] G. J. Myers. *The Art of Software Testing*. John Wiley & Sons, first edition, Feb. 1979.

[30] S. C. Ntafos. On random and partition testing. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis 1998 (ISSTA'98)*, pages 42–48, Clearwater Beach, FL, USA, Mar. 1998. ACM Press.

[31] S. C. Ntafos. On comparisons of random, partition, and proportional partition testing. *IEEE Transactions on Software Engineering*, 27(10):949–960, Oct. 2001.

[32] Symantec Antivirus Research Center. Expanded threat list and virus encyclopedia. Published online at http://securityresponse.symantec.com/avcenter/venc/data/cih.html. Last accessed: 16 Jan. 2004.

[33] P. Ször and P. Ferrie. Hunting for metamorphic. In *Proceedings of 2001 Virus Bulletin Conference (VB2001)*, pages 123 – 144, September 2001.

[34] TESO. Burneye ELF encryption program. Published online at http://teso.scene.at. Last accessed: 15 Jan. 2004.

[35] The WildList Organization International. Frequently asked questions. Published online at http://www.wildlist.org/faq.htm. Last accessed: 16 Jan. 2004.

[36] Virus Bulletin. VB 100% Award. Published online at http://www.virusbtn.com/vb100/about/100use.xml. Last accessed: 16 Jan. 2004.

[37] C. Wang. *A security architecture for survivability mechanisms*. PhD thesis, University of Virginia, Oct. 2000.

[38] West Coast Labs. Anti-virus Checkmark level 2. Published online at http://www.check-mark.com/checkmark/pdf/Checkmark_AV1.pdf. Last accessed: 16 Jan. 2004.

[39] West Coast Labs. Anti-virus Checkmark level 2. Published online at http://www.check-mark.com/checkmark/pdf/Checkmark_AV2.pdf. Last accessed: 16 Jan. 2004.

[40] E. J. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, July 1991.

[41] G. Wroblewski. *General method of program code obfuscation*. PhD thesis, Institute of Engineering Cybernetics, Wroclaw University of Technology, Wroclaw, Poland, 2002.

[42] z0mbie. Automated reverse engineering: Mistfall engine. Published online at http://z0mbie.host.sk/autorev.txt. Last accessed: 16 Jan. 2004.

[43] z0mbie. z0mbie's homepage. Published online at http://z0mbie.host.sk. Last accessed: 16 Jan. 2004.